University of Alberta

Viewing Immersive Images Using Virtual Cameras

by

Kenneth P. Der

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1996

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-18251-7

Canada

University of Alberta

Library Release Form

Name of Author: Kenneth P. Der

Title of Thesis: Viewing Immersive Images Using Virtual Cameras

Degree: Master of Science

Year this Degree Granted: 1996

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Kenneth P. Der
#206 9258 110A Ave.
Edmonton, Alberta
Canada, T5H 1J4

Date: Oct. 4 1996

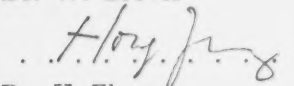
University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Viewing Immersive Images Using Virtual Cameras** submitted by Kenneth P. Der in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. A. Basu

Dr. W. Grover



Dr. H. Zhang

Date: 10/1/96

Abstract

We have developed a software system that allows users to interactively view immersive images. The immersive images capture large fields of view, typically 360° around for panoramic and panospheric images, with vertical fields that vary from 70° to complete spheres. The input images were captured using various imaging sensors including panoramic cameras, conic mirrors, and fish-eye lenses. These images are inherently distorted due to the non-planar projections involved. The software simulates reality by mapping the captured image onto a 3D model, such as cylindrical for panoramic images and spherical for fish-eye and panospheric images, and then allow the user to interactively view the scene with the aid of a virtual camera. We leveraged technologies such as image compression, computer graphics, and object oriented technologies such as Active X to realize a software implementation which targets the huge multimedia market.

Acknowledgements

I would like to thank my supervisor, Dr. Anup Basu, for his general guidance and support. He gave me a lot of latitude to explore many interesting new technologies, fulfilling my personal objective of being knowledgeable in many areas and not so much just an expert in one thing. His connections with industry, PVSI in particular, has provided me with the opportunity to work on this panospheric imaging software.

I would like to thank the thesis committee members for taking the time to read my thesis and for the helpful suggestions and constructive criticism.

I would like to thank Mr. Jerry Reyda, president of PVSI, for his encouragement and support. His business savvy and vision has created many opportunities for me at PVSI. JR manages to make routine meetings interesting and fun. The golf games at the JR Golf course was a welcome break from the hectic work schedule. The business lunches were great too, especially all that dessert!

Steve Bogner, at Piercorp, provided the fish-eye images of Ottawa and Dave Southwell, at PVSI, provided the panospheric images. Their images and helpful discussions about panospheric technology are appreciated.

The U of A computing science support staff were helpful, especially Steve Sutphen. Their help is appreciated.

I appreciate Kevin, Warren, and Mark's help with LaTex and unix stuff. Their crash course on LaTex give me a running start in the preparation of this thesis. Warren was also helpful in proofreading part of my thesis and beta testing my software. All of their cool music CDs kept the vision lab a happening place. (Although most of Kevin's collection was from my musical dark age)

I would like to thank Anne Nield for taking the time to proofread my thesis and for suggesting improvements to make this work more readable.

Sing Kwok Cheng was very helpful in suggesting the organization of part of my

thesis. His *ntalk cheng@nyquist.ee.ualberta.ca* sessions relieved some of the monotony of all those late night thesis writing sessions.

Last but not least, I would like to thank my family and friends for their support and encouragement, especially my parents, without whom none of this would be possible.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Applications 2

1.3 Thesis Organization 4

2 Background **5**

2.1 Image Processing Fundamentals 5

2.1.1 Sampling and Filtering 5

2.1.2 Image Compression 8

2.2 3D Graphics Fundamentals 10

2.2.1 Affine Transforms 10

2.2.2 Coordinate Systems 12

2.2.3 Camera Model 13

2.2.4 Texture Mapping 15

2.2.5 Quadric Surface 15

2.2.6 Line Sphere Intersection 16

2.2.7 Line Cylinder Intersection 19

2.2.8 Circle and Minimal Sphere from 3 Points 20

2.2.9 Graphics Pipeline 21

2.3 Previous Work 22

2.3.1 Field of View 22

2.3.2 Mosaicing 25

2.3.3 Panoramic Lens 26

2.3.4 Conic Mirrors 26

2.3.5 Panoramic Imaging Software 27

2.4 Methodology 29

3 Requirements **32**

3.1 Recommended Minimal Target Platform 32

3.2 Viewing with a Virtual Camera 33

3.2.1 Panoramic Images 33

3.2.2 Fish-eye Images 36

3.2.3 Front and Back Fish-eye Hemispheric Image Pairs 36

3.2.4 Upper and Lower Hemispheric Image pairs 38

3.2.5 Panospheric Images 39

3.3 Image Acquisition 40

3.4 User Interface 41

3.5 Technical and Economic Feasibility 43

4 Design **45**

4.1 Choice of Platforms 45

4.1.1 Operating System 45

4.1.2 Programming Language 46

4.2 Design Considerations 47

4.2.1 Software Re-usability and Performance 47

4.2.2 User Interface 48

4.3 Software Architecture 48

4.3.1 Document/View Model 49

4.3.2 Message Dispatching 51

4.4 File and Texture Image Format 53

4.5 Object Space to Texture Space Mapping 55

4.6 Integration with Other Applications 63

4.6.1 Component Object Model - Active X 63

5 Implementation **65**

5.1 Speed Considerations 65

5.2 Memory Considerations 67

5.3 Debugging 67

5.3.1 Assert the world 68
5.3.2 Visual Aids for Debugging 68
5.4 Matching Image Seams 71

6 Results **72**
6.1 Panoramic Images 72
6.2 Front and Back Fish-eye Image Pairs 73
6.3 Input Image Resolution 73
6.4 Panospheric Image Mapped to a Panoramic Image 76
6.5 Windows 95/NT 76
6.6 Image Compression 76
6.7 Active X control 79

7 Discussion **82**
7.1 Performance Evaluation 82
7.2 Advantages of Panospheric Imaging 82
7.3 Future Work 84
7.3.1 Panospheric Video 84
7.3.2 Automation of Preprocessing 85
7.3.3 Integration with other Applications 85
7.3.4 Head Mounted Display 86
7.3.5 Internet Applications 86

8 Conclusion **87**

Bibliography **89**

A C++ Fixed Point Number Class **91**

List of Figures

2.1 Signal sampling in time and frequency domain 6
2.2 Sampling techniques 7
2.3 Storage requirements for MIP map 8
2.4 Coordinate Systems 13
2.5 Camera Projection Types 14
2.6 Camera Coordinates 14
2.7 Control points of a Quadratic Surface 16
2.8 Line and Sphere Intersection 18
2.9 Radial Line and Sphere Intersection 18
2.10 Cylinder 19
2.11 Cylinder and Radial Line Intersection 20
2.12 Circle given 3 points 20
2.13 Graphics Pipeline 22
2.14 Panospheric Field of View 23
2.15 Field of View Map 23
2.16 Panoramic Field of View 24
2.17 Panoramic Field of View Map as a function of angles in spherical coordinates 25
2.18 Field of View Distortion 26
2.19 Panoramic Lens 27
2.20 Conic Mirror Field of View 28

3.1 Panoramic Images 34
3.2 Panoramic Distortion (a) Top view of cylinder (b) Plot of distance to the wall 34

3.3 Height of Wall Projected onto a cylindrical surface 35
3.4 3D Model of Panoramic Images 35
3.5 Panoramic Images 36
3.6 Fish-eye image of Ottawa 37
3.7 Front and Back Hemispheric Model 37
3.8 Front and Back Fish-eye Panospheric Image Pair (Ottawa) 38
3.9 Upper and Lower Hemispheric Image pairs 38
3.10 PVSI's Panospheric Optic 39
3.11 3D Panospheric Model and the corresponding image plane 39
3.12 Sample Panospheric Image 40
3.13 Cost of Various Panoramic Image Acquisition Technologies 44

4.1 Multiple Document Interface 49
4.2 Multiple Document Interface Screen Shot 50
4.3 Document View Model 52
4.4 MFC Message Dispatch Mechanism 52
4.5 PVS file format 54
4.6 Image Storage format 55
4.7 Zooming using a virtual camera 56
4.8 Camera Direction 56
4.9 Active Window regions for Camera movement 58
4.10 Property sheet for Camera Parameters 58
4.11 QPatch Control Points in Viewport Coordinates 59
4.12 QPatch Control Points in Object Model Coordinates 60
4.13 Panoramic Strip Viewing Model 61
4.14 General principle in Panospheric Mapping 62
4.15 Resolution Distribution Model 62

5.1 Debugging Visual Aid 70

6.1 Input Panoramic Image 72
6.2 Virtual Camera View of Test Panoramic Image 73
6.3 Input Panoramic Image 74

6.4 Virtual Camera View of Panoramic Image 74
6.5 Views of Front and Back Fish-eye images 75
6.6 Different Resolution Source Images 77
6.7 Panospheric Image Mapped to a Panoramic Image 78
6.8 Windows 95 User Interface 78
6.9 Impact of JPEG Compressed Source Images 80
6.10 Screen-shot of Internet Explorer with a panoramic imaging Active X control 81

Chapter 1

Introduction

1.1 Motivation

The Panoramic Viewing System developed here is a computer program to enable the user to interactively view a scene in any direction with the use of a virtual camera. It allows the user to pan left or right, tilt up or down, and zoom in or out. This type of system falls into a category of Virtual Reality (VR) in which the goal is to recreate the real world as convincingly as possible — as opposed to the category of VR that tries to convince the users that they are in another reality of the computer's making. With this form of immersive imaging, we are using real world scenes that were captured with various sensors, and applying computer graphics and image processing techniques to correct the inherent distortions. The basic function provided by the software is to allow the user to view warped images of a scene while correcting the distortions. The images are warped because all the information in a 3D scene is packed into a 2D image; essentially, a non planar projection is being mapped to a planar projection. The software takes the 2D image and tries to undo the distortions. This unwarping is accomplished by supplying a 3D model that can be used in conjunction with the 2D images in order to reconstruct the scene, such that a virtual camera can be used to interactively look around as if we were inside the scene.

The motive for this project is to create economic benefits from a technology that will enable the users to interactively view immersive images of real world scenes on readily available inexpensive Personal Computers (PCs). The increase in processing power of the PC has made this technology feasible for the mass market. There is an explosion of interest generated by the Internet, more specifically the World Wide

Web, in multimedia content. The technology could become a valuable tool in the toolkits of multimedia content creators. At present, the Internet application of this technology for the majority of the users will not be very satisfying because of the long delays brought about by low bandwidth modem connections. As more high bandwidth technologies such as ATM and ADSL are deployed to the end users, this imaging technology will become more attractive for Internet applications. CD-ROM and Intranet based applications can apply the technology now because the transfer rates are not as limiting as modem connections.

Panoramic imaging technology can be applied in many areas where conventional static images may be inadequate, or where synthetic computer models would be difficult or time consuming to build. A major problem with conventional Virtual Reality systems is the labour intensive process of building models. Computer models of a real world scene would not only be labour intensive to build, but the rendering of the model would also be computationally intensive. The technology we have developed allows the use of a simple generic 3D model and maps the captured image of the scene onto that model, thus not only simplifying the model building process, but also decreasing the computational complexity of the rendering process. This method does not involve extracting 3D models from static images and then manipulating and rendering the model. Feature extraction and classification is a non-trivial problem and is the subject of much on-going research, but we need not deal with this problem since we supply a 3D model implicitly.

1.2 Applications

There are many potential applications of panospheric imaging, some of which are listed below.

- Real Estate Visualization - With Panospheric images the home buyer could view many real estate sites on the computer and have the ability to look around. This process would save time since it could be used to eliminate properties that they were not interested in.

- Scene Capture - Panospheric images can be used to capture crime scenes or traffic accident scenes — if not in detail, then to provide context for the other photographs. Basically, it could be used to give the spatial relationships of the various objects in the scene. These panospheric images could also be used for insurance purposes to catalogue objects of value.

- Video Games - There is a huge video games market for the home computer. The increasing processing power of entry level PCs enables video game developers to design more and more realistic video games. Panospheric images could be one of the ways to add realistic backgrounds to certain types of video games. Panospheric images have the advantage of being able to capture details that may be very difficult or time consuming to simulate.

- Medical Imaging - Panoramic imaging systems can be used to view panoramic dental x-rays. This may be of questionable value in practice because dentists may already be used to looking at the panoramic x-rays with the distortions.

- Astronomy - Panospheric imaging could be used to capture views of the night sky. Because of the large field of view, it could capture the entire sky in a single photograph, as well as the portions of the earth which would give context to the scene.

- Virtual Guided Tours / Tourism - For instance, panospheric images could be taken of various locations in a city. Software could be written to show a map of the city and, as the user selects various locations on the map, a panospheric view would be presented allowing them to look around. Such an application would be extremely useful for events that attract a lot of visitors, such as the Olympics. Using this software tourists could familiarize themselves with the locations that they are interested in.

- Multiple Panospheric Images for triangulation - Triangulation could be used to determine distances of objects in the scene. This could be used for reconstruction of a scene (of traffic accidents for instance)

- Warping to Omnimax format - Panospheric images have a larger field of view than Omnimax, therefore it is possible to warp panospheric images to Omnimax format. It is therefore conceivable that a single imaging system could be used to capture Panospheric data, and to generate (post processing) Omnimax frames.

- Multimedia Education systems - This imaging system could be used in situations where a large field of view could enhance understanding, and is limited only by the imagination of the content creator.

1.3 Thesis Organization

We start by reviewing some of the fundamental concepts required for a thorough understanding of this work. Some concepts, such as 3D graphics, are essential while others — such as filtering, sampling, and compression — are not as essential. An overview of previous related work will be presented, including both hardware and software panoramic image acquisition and transformation systems. We will then examine the methodology used in the development of the software, which will be loosely based on software engineering principles. We next specify the requirements for this project, followed by some analysis of the requirements. The design procedures and decisions will be presented, followed by a discussion of the implementation issues. The results of the project will be presented and discussed. Finally, a discussion of future work and unresolved issues will be presented, followed by the conclusion. Whenever possible, the material is presented in increasing levels of detail with the general concepts being presented first followed by a more technical treatment of the topic for the interested reader.

Chapter 2

Background

Here we introduce some basic graphics and image processing concepts that will be useful for understanding the work in the remainder of this thesis. In this chapter, we will not concern ourselves with how the concepts and ideas are used in the project; we will present the concepts by themselves, and refer to them again in the design and implementation phase.

2.1 Image Processing Fundamentals

In this section we will review some of the fundamental concepts, such as sampling and filtering, where an essentially continuous signal is sampled and represented by a discrete number of values. An approximation of the original signal is then reconstructed from the sampled and filtered signal. Since we will be reconstructing and resampling the images when we map them onto 3D models, an understanding of basic sampling and filtering concepts will be helpful. A brief overview of image compression will also be presented, since it is used to reduce the storage requirement of the input images.

2.1.1 Sampling and Filtering

The concept of transforming signals from one domain to another is well known and has been used extensively in signal processing (and image processing). An important concept is that a signal can be represented as a linear combination of basis functions. It can be shown that a signal (in 1D) can be transformed into a frequency representation where there are a set of basis functions and corresponding weighting factors. Similarly an image (in 2D) can be transformed into a frequency representation. It

can also be shown that, theoretically, a signal can be reconstructed if the sampling frequency is at least twice that of the highest frequency component in the original signal. This is assuming that we have an ideal reconstruction filter, which is almost never the case in graphics; therefore, the sampling frequency in practice needs to be higher than twice the highest frequency component. In 1D, sampling can be thought of as multiplying a signal $f(t)$ by an impulse train, sometimes called a comb function, $s(t)$ in the time domain as shown in Figure 2.1. The resultant signal is the sampled signal $f_s(t)$. The multiplication in the time domain corresponds to a convolution op-

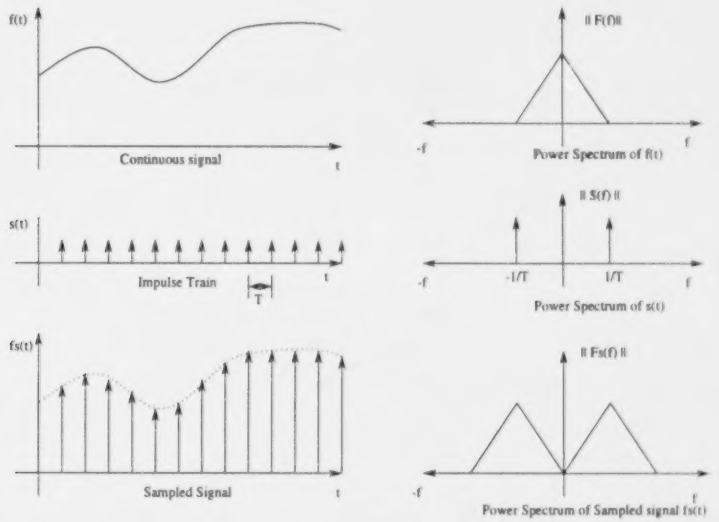


Figure 2.1: Signal sampling in time and frequency domain

eration in the frequency domain. The interesting characteristic of the impulse train in the time domain is that it is also an impulse train in the frequency domain. If the spacing between the impulses is T in the time domain then the spacing is $1/T$ in the frequency domain. This makes sense, since T and $1/T$ are the period and frequency of the impulse train. Note that if the sampling frequency is less than twice that of the highest frequency component, the resultant convolved signal will have overlapping frequency components. When the signal is reconstructed the overlapped frequencies are no longer separable and masquerade as lower frequency components; this is what is referred to as *aliasing*. We could prevent aliasing by first filtering out the high frequency components before sampling, or we could increase the sampling frequency.

Image filtering can be used to improve image quality. When mapping images

from one space to another, the pixels in the destination may not fall exactly on texel[1] (or source) coordinates. The easiest, or computationally least expensive way, to handle this situation is to find the nearest neighbour and use that value. This is essentially a resampling of the image based on a zero order hold reconstruction. Another approach, which is slightly more expensive, is bilinear interpolation. Bilinear interpolation uses the four surrounding texels and interpolates the desired value based on the distances involved. Since the resampling is based on a better reconstruction, because it uses the four surrounding points, the results will be better. Figure 2.2 illustrates both of these methods. There are more elaborate filtering techniques,

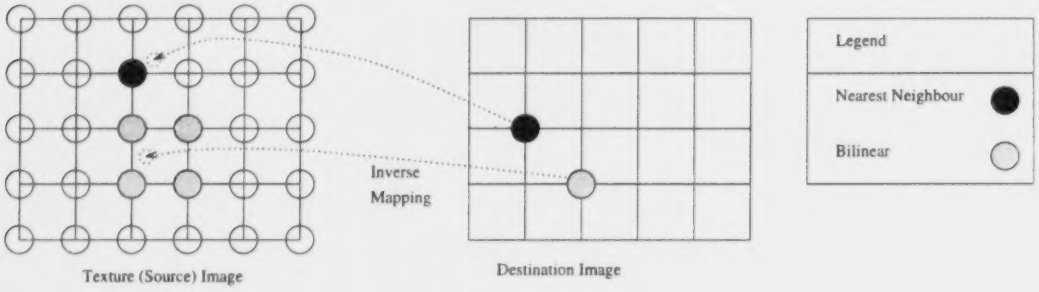


Figure 2.2: Sampling techniques

such as elliptical weighted average (EWA) filtering [8], which would be prohibitively expensive for interactive systems on most PCs. The EWA method projects a circular pixel onto the object surface which becomes an ellipse. All the texels within the ellipse are Gaussian weighted to give a final value. Adaptive supersampling and MIP mapping are other techniques that can be used. The basic idea of MIP mapping is to precompute successively smaller versions of the texture images such that the next level is half the height and width of the previous image, or 1/4 of the area. The memory requirement of a mipmap representation of a texture image is 4/3 that of the memory required by the original texture image, as illustrated by Figure 2.3. The successively smaller levels of the mipmap (lower right quadrant) are essentially prefiltered (low pass filtered) versions of the higher level image. Since mipmaps can be precalculated, the filtering operation is not too expensive computationally.

[1]texel refers to a pixel in the texture image.

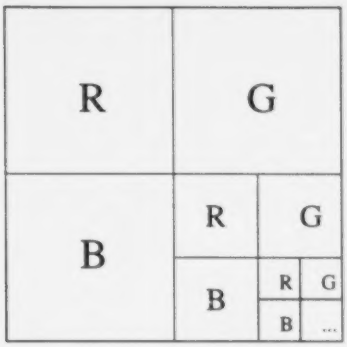


Figure 2.3: Storage requirements for MIP map

2.1.2 Image Compression

Image compression [7, 29] is a large field in itself and a thorough treatment is beyond the scope of this thesis. We will merely review the general concepts behind image compression. The goal of compression is to represent information in such a way as to reduce the storage requirements. In data compression there is a distinction between *data* and *information*. Different amounts of data can be used to convey the same amount of information. This thesis, for instance, can be compressed using various compression utilities or, on a slightly higher level, compression can be achieved by reorganizing and rewriting the thesis in a more concise way. In this example, the information is the ideas presented in this thesis, and not the words used to express these ideas. The words would be the data, therefore if fewer words are used to convey the same amount of information, then compression is achieved. The concept of data compression is nothing new and predates the modern computer. The use of shorthand and contractions are examples of data compression.

Image compression is an attempt to reduce data redundancy[7].

- **Coding Redundancy** occurs when fewer code symbols can be used to represent the data stream. Techniques such as Huffman coding or arithmetic coding can reduce coding redundancy by assigning shorter codes to the most probable symbols.

- **Interpixel Redundancy** occurs when the neighbouring pixels can be predicted to a reasonable degree. This implies that there is very little new information.

- **Psycho-Visual Redundancy** occurs because of the limitations of the human visual system. For instance, we are more sensitive to differences in light intensity than color intensity.

Image compression techniques usually incorporate methods to reduce some or all of the above categories of data redundancy. Image compression is important because of the massive amounts of data involved. For instance a 1024×1024 24 bit image would require 3 megabytes of storage, if left uncompressed. This would not only tax the secondary storage, such as hard drives, but transfers across networks would be slow. But, Image compression is not free. It comes at the expense of encoding and decoding the images, which takes CPU (Central Processing Unit) time. This usually is not a major concern since there are often many idle cycles on most machines. There are two categories of image compression incorporated in countless graphics file formats [31]: lossless and lossy.

Lossless Compression

With lossless compression the input image is completely recoverable from the compressed data. Standards such as Graphics Interchange Format (GIF)[2] and Portable Network Graphics (PNG)[3] are examples of common lossless formats for storing images. The advantage of lossless compared to lossy compression is that it does not alter the image, so what you put in is what you get out. This comes at the expense of not being able to compress the image as much as lossy compression methods. In lossless compression, computer generated images can achieve much higher compression ratios than images of natural scenes, in most cases. Although GIF is considered a lossless compression method, it can only handle 8 bit images giving a maximum 256 colors. Therefore, all 24 bit images are mapped to an 8 bit palette first, thereby achieving a 3 to 1 lossy compression before the lossless LZW [31] compression is performed.

[2]GIF is a standard image file format that was proposed by a group of engineers from various software companies, and sponsored by Compuserve, back in 1987. In December 1994, Compuserve announced that the LZW compression routine used in the GIF format infringed a Unisys patent. As a result, Unisys now requires a royalty for software supporting the GIF format.

[3]PNG was developed to address the weaknesses of the GIF format and to be unencumbered legally.

Lossy Compression

Possibly the most widely used lossy compression format is JPEG[18, 32]. It uses the Discrete Cosine Transform (DCT) on 8x8 image sub-blocks and quantizes the DCT coefficients, which takes into account psychovisual effects. The compression is the result of many zero DCT coefficients after the quantization which can then be run-length encoded. The resultant data stream is then Huffman encoded to reduce coding redundancy. JPEG does a reasonably good job of compressing real world or photographic scenes.

Although other lossy compression algorithms exist, and in many cases give better results, we will use JPEG because it is widely supported on virtually all computer platforms. With lossy compression we can usually choose between good quality and good compression. Often the quality loss cannot be perceived, even at a relatively high compression ratio such as 20 to 1.

2.2 3D Graphics Fundamentals

In this section we will introduce some of the 3D graphics fundamentals that were used to implement the software. We will start by presenting the basic affine transforms that can be used to manipulate the other objects, which include virtual cameras, coordinate systems, and geometric primitives. The general concept of texture mapping will be presented, followed by a brief explanation of quadric surfaces. We will then examine how some basic line/surface intersections can be calculated. With this knowledge in hand, we will see how a graphics pipeline ties all this together. An understanding of 3D graphics is important because we will be modeling and transforming the warped input images as 3D objects.

2.2.1 Affine Transforms

The characteristic of affine transforms that we are most interested in is the fact that straight lines remain straight and parallel lines remain parallel after affine transformations. Loosely speaking, affine transforms make sense geometrically. For instance, one can transform the endpoints of a line; drawing the line from the transformed endpoints will be the same as drawing all the points in the line first, then transform-

ing each of those points. This also means that we can apply these transforms on a set of control points for geometric primitives. Affine transforms include translation, rotation, scaling, and shearing operations, but we will only concern ourselves with the first three.

Translation

Translation allows us to move a geometric primitive from one location to another. More precisely, it adds a displacement vector to a point in affine space. The translation matrix is given by Equation 2.1.

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

Rotation

Rotation allows us to rotate a geometric primitive about a given axis. It can be shown that rotation about an arbitrary axis can be implemented as a series of rotations about the primary axes. This transform will be important for transforming the virtual camera to look at other regions in the virtual scene. The rotation matrices about the z, x, and y primary axes are given by Equations 2.2, 2.3,and 2.4, respectively.

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Scale

Scaling allows us to change the size of the geometric primitive with respect to the origin. The scaling matrix is given by Equation 2.5, where s_x,s_y, and s_z are the scaling factors in the x,y, and z directions, respectively.

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Compound transformations can be performed by multiplying all the individual transformation matrices together. This allows us to perform complex transformations on many data points without having to do all the matrix multiplications for every data point. After the net transformation is computed, each of the data points can be transformed using a single matrix multiplication.

2.2.2 Coordinate Systems

An important notion is that of using multiple coordinate systems. One way to organize a coordinate system is to have a global coordinate system, which we will call *world coordinates*. All other coordinate systems are relative to the world coordinates. The camera can be manipulated using any of the affine transforms. All geometric objects have, and are defined in, their own coordinate systems. They are then modified by affine transforms in the world coordinate system. Figure 2.4 shows a world coordinate system with one object coordinate system and one camera coordinate system defined relative to it. The object coordinate system could in turn contain sub-objects. In other words, we can setup a hierarchy of relative coordinate systems. We could use translation operations to move the camera to different locations within the world coordinate system, and apply rotations to point the camera in the desired direction. The objects can be moved, rotated, and resized by applying the translate, rotate, and scale operations respectively. Although this concept is useful, for our particular problem we can use a simplification by locating the camera and the geometric object at the origin of the world coordinate system. Note that there is no restriction on the number of cameras that the world coordinate system can contain, thus allowing us to look at the same scene from different perspectives through the different cameras.

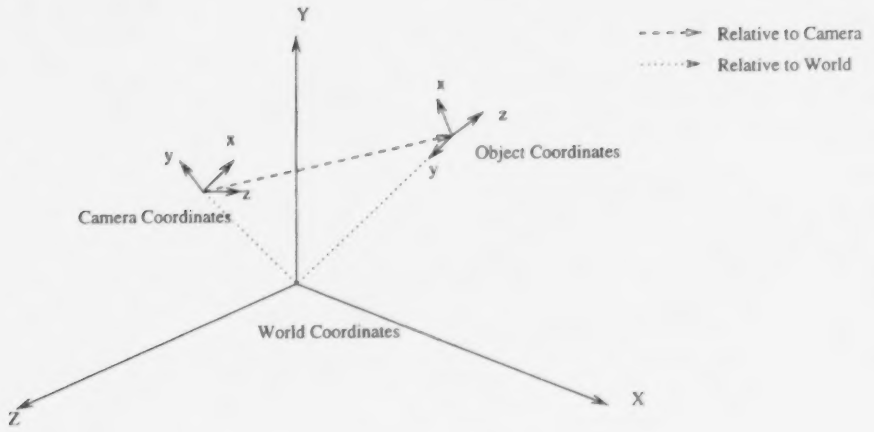


Figure 2.4: Coordinate Systems

To see what the camera sees, we first need to find the transform that transforms the object from object coordinates to world coordinates. Next, we transform the world coordinates to the camera coordinates. Now all the objects are relative to the camera coordinates. Clipping and rendering can now take place.

2.2.3 Camera Model

There are two types of camera projections commonly used in computer graphics, as shown in Figure 2.5: perspective and parallel (sometimes call orthographic) projection. As can be seen from the figure, the size of the object in the parallel projection is unaffected by the distance of the object from the camera. In the perspective projection, the distance does affect the size of the object as projected onto the projection or viewing plane. More specifically, the further the object is from the camera, the smaller the projection. This is the type of projection that most of us are familiar with because our visual system uses perspective projection.

Parallel projection is useful when displaying flat images that are coplanar to the viewing plane. Using perspective projection in that case would result in a slight distortion, since the pixels in the center of the view plane are closer to the camera than the pixels in the outer edge of the viewport.[4] The camera model is given in Figure 2.6, with the camera defined in its own coordinate system. In this camera viewing model, the near clip plane is actually the viewport, which in turn is usually

[4]A viewport is the visible, usually rectangular, region on the viewing plane.

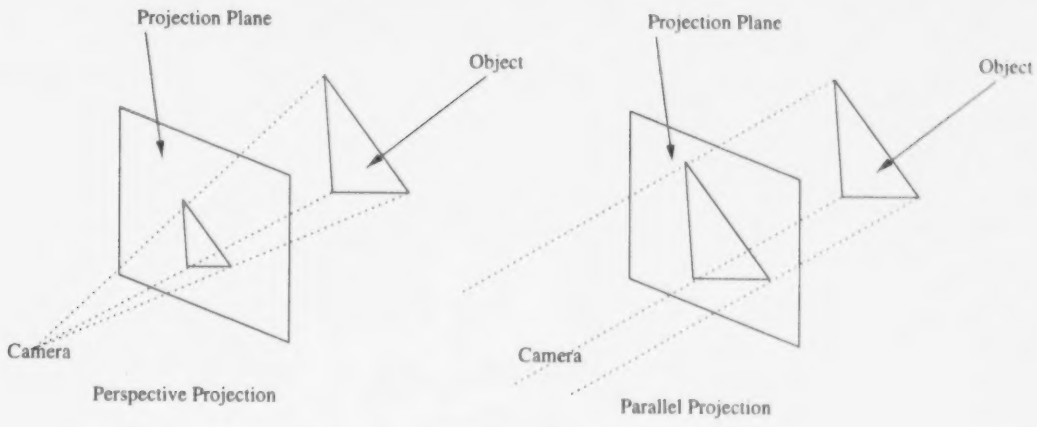


Figure 2.5: Camera Projection Types

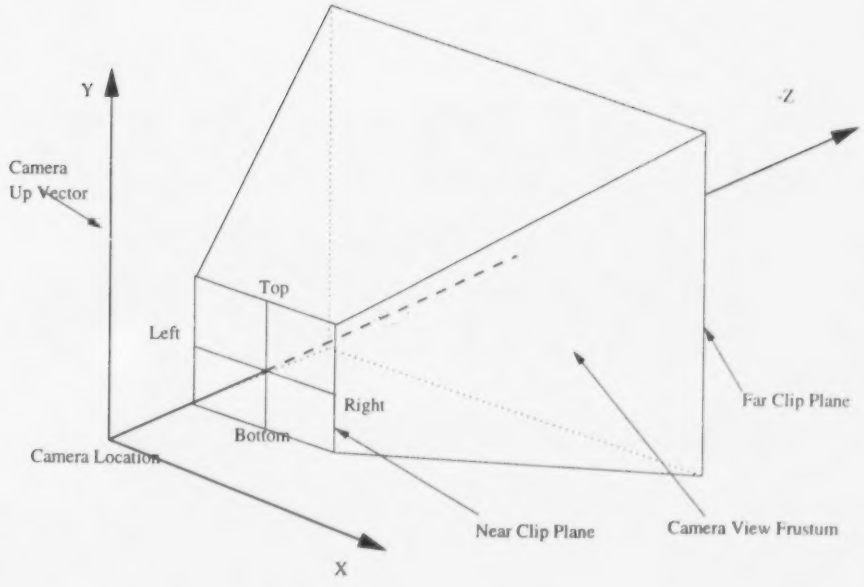


Figure 2.6: Camera Coordinates

mapped to a window. The focal point is located at the origin. The Y axis is the camera's up vector. The viewing frustum (sometimes referred to as the clipping volume) is defined as the pyramid volume between the near and far clip planes. This implies that any points outside the clipping volume will not be rendered.

2.2.4 Texture Mapping

Texture mapping is a well known computer graphics technique for mapping an image from texture space to object space [6]. The typical use is for simulating surface details on 3D objects. From the image processing point of view, this is an application of image warping in that an image is mapped from one coordinate space to another. Image warping, however, generally deals with the source and destination as 2D objects only. That is to say, image warping maps from one plane to another plane. Texture mapping is done using inverse mapping, where the destination pixel coordinate is used to determine the corresponding object coordinate. The object coordinate is in turn used to determine the corresponding texture coordinate. Basically, this is a resampling problem since not all the texture coordinates will be discrete integer values. The texture space is reconstructed and resampled as was discussed in the sampling and filtering section.

2.2.5 Quadric Surface

An understanding of curved surfaces, such as quadric surfaces, is important because we will be mapping our input images onto these curved surfaces in order to correct the distortions.

A quadric surface can be generated with the following equation:

$$f(x,y,z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0 \quad (2.6)$$

Note that a unit sphere can be represented if $a = b = c = -k = 1$ and all the other coefficients are zero. The quadric surface becomes a plane if the constants a through f are zero.

We can use a quadratic patch to approximate a quadric surface if we choose the nine control points to be on the surface of the quadric surface. The quadratic patch

is interpolated in the u and v directions [11] using the following equation:

$$f(u,v) = \begin{bmatrix} 2v^2 - 3v + 1 & 4v(1-v) & v(2v-1) \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} & P_{02} \\ P_{10} & P_{11} & P_{12} \\ P_{20} & P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} 2u^2 - 3u + 1 \\ 4u(1-u) \\ u(2u-1) \end{bmatrix} \tag{2.7}$$

The control points are located in the interpolating (Catmull-Rom) basis [4]. The interpolating basis is a basis in which the surface corners will be at the corner control points, and the surface will pass through all control points, as shown in Figure 2.7. We

Figure 2.7: Control points of a Quadratic Surface

can quickly check this by using the combinations of $u = 0, 0.5, 1$ and $v = 0, 0.5, 1$. The resulting points are indeed the control points. The quadratic patch can be recursively subdivided by using Equation 2.7 to find the new control points. This can be done to a specified depth (of recursion), and then a forward differencing algorithm could be used to compute the points for the smaller patches, using fewer computations per point.

2.2.6 Line Sphere Intersection

Some of the 3D models that we will be using are spherical; therefore it is important that we cover basic line/sphere intersection calculations.

The general equation of a sphere is given by:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = R^2 \tag{2.8}$$

where $P(x_c, y_c, z_c)$ is the center and R is the radius of the sphere.

The parametric equation for a line in 3D space is given by:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + t \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} \tag{2.9}$$

The first step is to translate the center of the sphere to the origin and apply the same transform to the line.

The sphere equation 2.8 now becomes

$$x^2 + y^2 + z^2 = R^2 \tag{2.10}$$

The line equation 2.9 now becomes

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x_0 - x_c \\ y_0 - y_c \\ z_0 - z_c \end{bmatrix} + t \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} \tag{2.11}$$

Substitute the 3 independent parametric equations for x, y and z from 2.11 into 2.10 we get the following:

$$(x_0 - x_c + t\,dx)^2 + (y_0 - y_c + t\,dy)^2 + (z_0 - z_c + t\,dz)^2 = R^2 \tag{2.12}$$

Let $(x'_0, y'_0, z'_0) = (x_0 - x_c, y_0 - y_c, z_0 - z_c)$

$$(x'_0 + t\,dx)^2 + (y'_0 + t\,dy)^2 + (z'_0 + t\,dz)^2 = R^2 \tag{2.13}$$

Expanding and collecting the terms we get

$$(dx^2 + dy^2 + dz^2)t^2 + 2(x'_0\,dx + y'_0\,dy + z'_0\,dz)t + (x'^2_0 + y'^2_0 + z'^2_0) - R^2 = 0 \tag{2.14}$$

which is in the form

$$At^2 + Bt + C = 0 \tag{2.15}$$

where $A = (dx^2 + dy^2 + dz^2)$, $B = 2(x'_0\,dx + y'_0\,dy + z'_0\,dz)$ and $C = (x'^2_0 + y'^2_0 + z'^2_0) - R^2$

If $A = 0$, then $B = 0$. In this case the line degenerates to a point. If $C = 0$, then the point is on the sphere; otherwise the point is outside the sphere if $C > 0$, and inside if $C < 0$.

If we have a valid line (i.e. $A \neq 0$), then we could end up with 0, 1, or 2 intersections given by the quadratic formula.

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{2.16}$$

If $t > 0$, then the intersection is in the positive direction of the ray. If $t < 0$, then the intersection is in the negative direction of the ray. If $B^2 - 4AC < 0$, then the line does not intersect the sphere. In the cases where there is one intersection, then $B^2 - AC = 0$; if $t = 0$, then the line originates on the sphere and is perpendicular

to the surface normal at that point (see Figure 2.8(a)). Figure 2.8(b) depicts the case when $t > 0$. In the cases where there are 2 intersections, $B^2 - 4AC > 0$; if both solutions of t are positive we have the case in Figure 2.8(c). If one of the solutions of t is positive and the other t is negative, then we have the case depicted in Figure 2.8(d) where the line originates inside the sphere. If both the solutions of t are negative, then we have the case depicted in Figure 2.8(e) where the ray intersects the sphere along the negative direction.

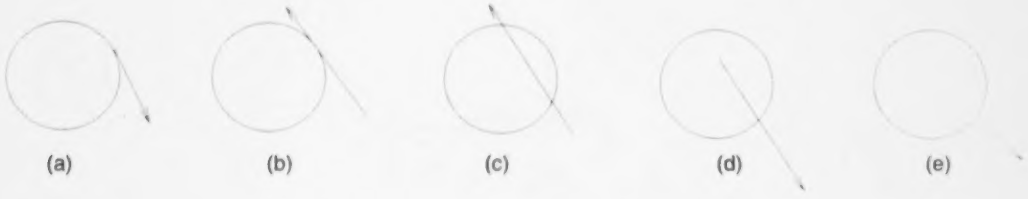


Figure 2.8: Line and Sphere Intersection

Although all these cases are interesting, we will mainly be concerned with the case depicted in Figure 2.8(d). More specifically, if we are only concerned with the special case where the ray originates from the center of the sphere, and we were given the direction of the ray as in Figure 2.9, then we could find the intersection point very efficiently by simply normalizing the direction vector — assuming the center of the sphere is at the origin and that it is a unit sphere. Otherwise, the unit direction vector is multiplied by the radius and then translated by the offset of the center of the sphere.

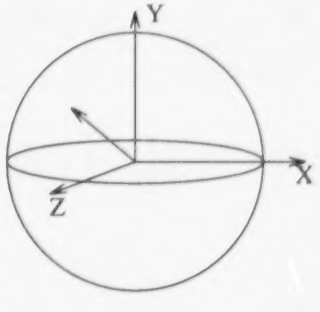


Figure 2.9: Radial Line and Sphere Intersection

2.2.7 Line Cylinder Intersection

Panoramic images are modeled as cylindrical surfaces, therefore we will present line cylinder intersection calculations.

The general approach to solving for the intersection of a line and a cylinder is very similar to that of solving for the intersection of a line and sphere. The first step is to transform the cylinder in question into a standard form, such as in Figure 2.10, where the cylinder is along the Z axis and with a unit radius. We then apply the same transform to the line. Now the problem becomes simply a case of line circle intersection in 2D (i.e., ignoring the z component). Once the intersection points are found the corresponding z values are computed. If the values are within the range of the z value for the cylinder, then there is a valid intersection; otherwise there is no intersection. The equations for the generalized case of line cylinder intersection

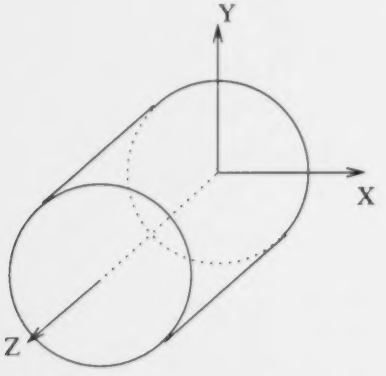


Figure 2.10: Cylinder

as described above will not be presented because there is a much more efficient way of solving for the special case that we are concerned with. The special case is shown in Figure 2.11, where the line starts at the origin and is in the direction of vector $\vec{v}$. To solve for the intersection, we first project vector $\vec{v}$ onto the XZ plane and find the length of the projected vector. By multiplying $\vec{v}$ by the reciprocal of the length of the projected vector, we are basically scaling $\vec{v}$ such that its projection on the XZ plane will be of unit length. Another way of looking at this is that we are essentially solving the similar triangle problem in 3D.

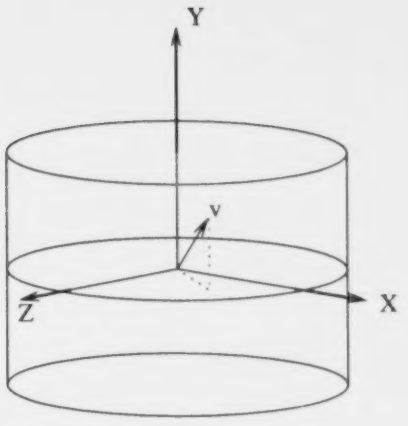


Figure 2.11: Cylinder and Radial Line Intersection

2.2.8 Circle and Minimal Sphere from 3 Points

In this section we will derive a procedure for finding a circle, given 3 points on the circle. This is equivalent to finding the smallest sphere that could contain the three points in 3D. In Figure 2.12 the three points on the circle are labeled as P_1,P_2, and P_3. The midpoint between P_1 and P_2 is labeled as P_{12}; similarly, P_{23} is the midpoint of P_2 and P_3. d_{12} and d_{23} denote vectors originating at P_{12} and P_{23}, respectively, and pass through the center of the circle.

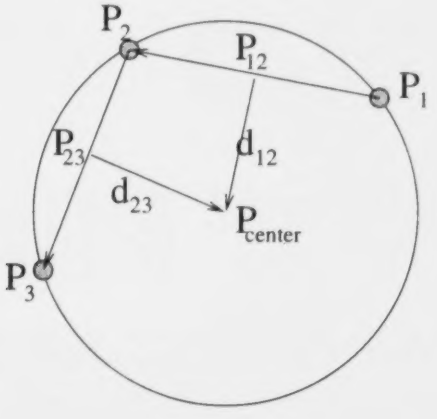


Figure 2.12: Circle given 3 points

First we note that any three non collinear unique points define a plane, and the Normal vector $\vec{N}$ — or the vector perpendicular to the plane — can be calculated by taking the vector cross product of non collinear vectors on that plane, such as $P_1\vec{P}_2$

and $\vec{P_1P_3}$.

$\vec{P_1P_2} \times \vec{P_1P_3} = \vec{N}$

With a normal vector, we can find a vector that is perpendicular to both $\vec{N}$ and the vectors on the plane. Where $\vec{d_n}$ represents a direction vector that is collinear with the vector defined by the center and the midpoint, then

$\vec{N} \times \vec{P_1P_2} = \vec{d_{12}}$

$\vec{N} \times \vec{P_2P_3} = \vec{d_{23}}$

We can find the midpoints P_{12} and P_{23} of the two vectors as follows

$P_{12} = (P_1 + P_2)/2$

$P_{23} = (P_2 + P_3)/2$

Now we can write an equation that states that if we have two points, and have corresponding vectors that are known to point toward the center, the intersection will be the center.

$P_{12} + t_1 \vec{d_{12}} = P_{23} + t_2 \vec{d_{23}}$

We can solve this equation because there are at least two unique equations, one each for the x, y, and z components, but only two unknowns (t_1 and t_2). Once we solve for t_1, we can simply find the center by using the following equation:

$P_{12} + t_1 \vec{d_{12}} = P_{center}$

Now we have the center of the circle or sphere and at least one point on the circle; it is simple to calculate the radius R:

$R = \|P_{center} - P_1\|$

2.2.9 Graphics Pipeline

A graphics pipeline is a sequence of operations that is performed to transform geometric objects into a display buffer to be drawn on the screen. The camera parameters, such as location, direction, and field of view, determine where the geometric objects should be on the screen. Figure 2.13 shows an overview of a graphics pipeline. The application supplies the camera parameters, such as the location, orientation, clipping volume, and type of projection. The application also supplies the geometric models and transforms them in the world coordinate system. The graphics pipeline takes the geometric objects and transforms them into the camera coordinate system; it can

then apply the clip volume transform. All objects that are outside of the clip volume are not rendered. The remaining objects are scan converted in the rasterization module. The resulting buffer is displayed to a window.

Figure 2.13: Graphics Pipeline

2.3 Previous Work

2.3.1 Field of View

The field of view for panospheric images is described as the percentage of the sphere surface for which the captured image represents, as described by Bogner [3]. It was assumed that the field of view would be circular for all panospheric images. We will generalize this and use it to describe the field of view for all camera models — including panoramic and panospheric, as well as conventional and wide angle cameras. We will simply define the percent field of view as the ratio of the captured area on the surface of the sphere compared to the total surface area of the sphere. That is to say, we will relax the restriction of circular regions by allowing non circular regions as well.

Panospheric Field Of View

Panospheric field of view can be calculated by taking the ratio of surface area in the field of view on a sphere to the total surface area of the sphere. The percent field of view, as a function of the angle from the vertical axis, is plotted in Figure 2.14(a) and is calculated as

$$\%Field = 100 \times \frac{\theta(1 - cos\alpha)}{4\pi} \tag{2.17}$$

In this case, if we assume circular regions or $\theta = 2\pi$ then the $\%Field = 50(1 - cos\alpha)$. Figure 2.14(b) illustrates the region involved; as the angle α increases, the size of the *cap* increases and the field of view increases correspondingly. As we cross the horizontal plane, the rate at which the field of view increases becomes smaller and corresponds to the inflection point in the function plot. As we can see, a 50% field

of view corresponds to the upper hemisphere, or exactly half the sphere. Figure 2.15 is a surface plot of Equation 2.17, where the restriction of circular regions has been removed so that θ can vary from 0 to 2π. As we can see, the function in Figure 2.14(a) is represented on this surface plot at the right hand edge where the horizontal angle or $\theta = 360°$.

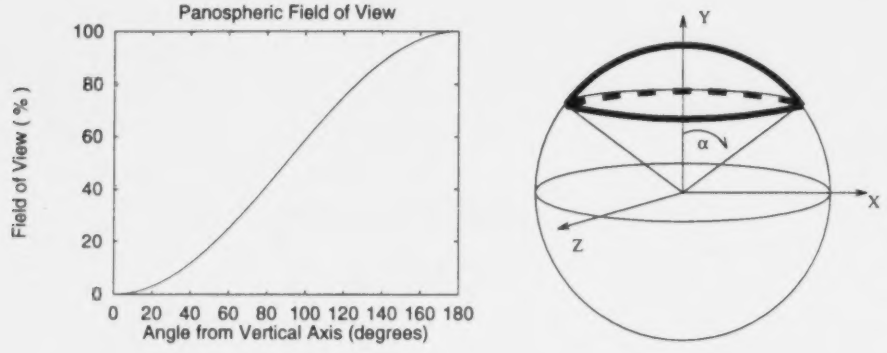


Figure 2.14: (a)Panospheric Field of View Function (b)Panospheric field of view corresponding to angle from vertical axis

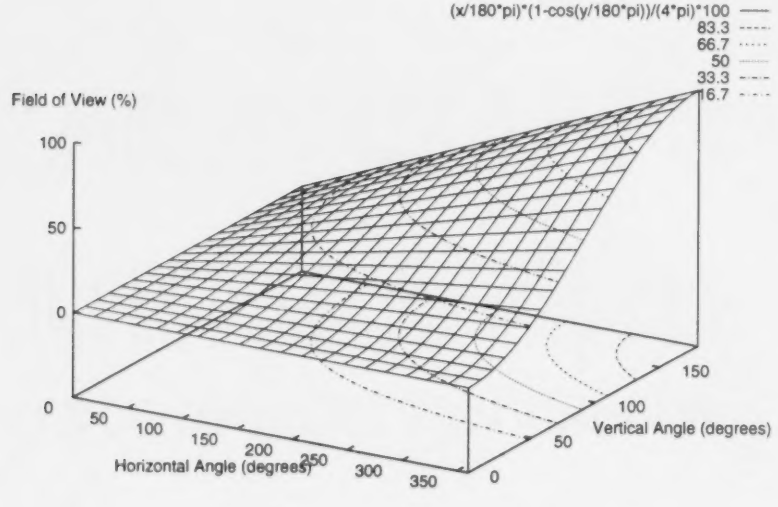


Figure 2.15: Field of View Map as a function of angles in spherical coordinates

Panoramic Field Of View

The field of view for panoramic images can also fit into this definition if we inscribe the cylinder in the sphere, such that the top and bottom of the cylinder are on the surface of the sphere and project the cylindrical surface onto the sphere. In Figure 2.16(b) the cylindrical surface is projected onto the sphere and is described as a function of the angle α which, basically, is a function of the ratio of the height and radius of the cylinder. The percent field of view for panoramic (or cylindrical surfaces) can be calculated as

$$\%Field = 100 \times \frac{\theta sin(\alpha/2)}{2\pi} \tag{2.18}$$

where $0° < \alpha < 180°$ and $0° < \theta < 360°$. Again, if we assume a complete 360°, or a complete cylinder, then $\theta = 360°$ which yields the plot in Figure 2.16(a). The surface plot for Equation 2.18 is given in Figure 2.17. As with the panospheric field of view surface plot, we can see the contours where there would be an equivalent field of view, but with differing α and θ angles. For instance, from the surface plot in Figure 2.17, a complete cylinder with a vertical field of 40° would require a half cylinder to have a vertical angle of roughly 100° to give a similar percentage field of view.

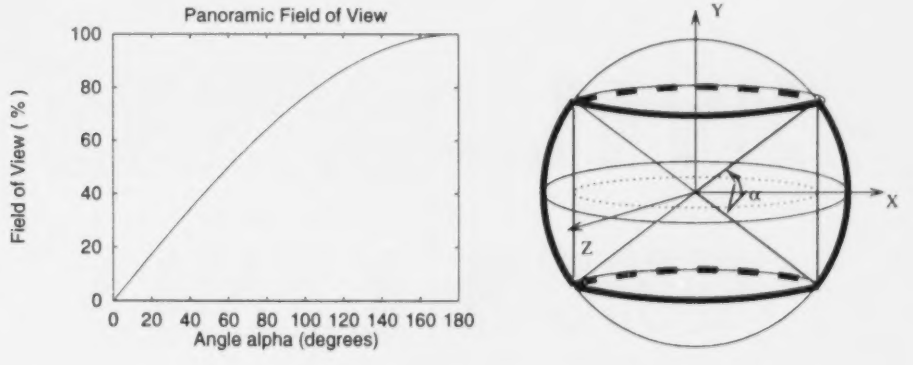


Figure 2.16: (a)Panoramic Field of View Function (b)Panoramic field of view corresponding to angle α

Distortions

When we store or capture panoramic or panospheric projections onto images, we are essentially mapping these non geometric projections onto a geometric projection such as the perspective projection. The distortions due to extremely wide fields of view can

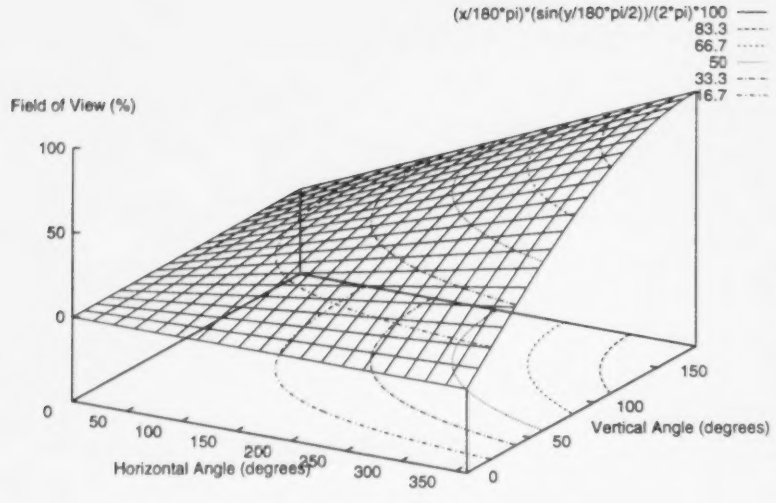


Figure 2.17: Panoramic Field of View Map as a function of angles in spherical coordinates

be thought of in much the same way as approximating a curved surface with a plane. In Figure 2.18 we can see that the amount of perspective foreshortening is dependent on the field of view of the perspective or conventional camera. The wider the field of view the more apparent the distortions. The basis for this project is to correct these distortions when we use a conventional camera model to view the images. We can see that the perspective foreshortening is more dramatic at the center of the viewing plane.

2.3.2 Mosaicing

Mosaicing is a process by which a sequence of smaller images is patched into a bigger image[25]. Panoramic images can be generated using this technique. It has also been used as a means of compressing video of a scene[13]. Omnimax images have been created from multiple perspective views using elliptical weighted filters [8].

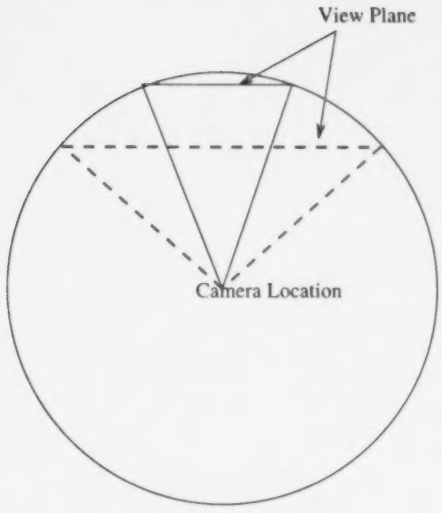


Figure 2.18: Field of View Distortion

2.3.3 Panoramic Lens

Panoramic images can be captured using a special lens designed to capture a panoramic field of view. Many such optical blocks or lenses exist, one of which is based on the optic block in Figure 2.19. The lens block designed by Powell [19] is capable of projecting a full 360° cylindrical field of view into an annular format. The light rays, represented by the dotted lines, enter the optic and undergo a total of two refractions and two reflections before exiting. This intermediate image is further transformed by a relay optic to produce the final annular image onto a two dimensional detector array. The concepts are interesting, and presented only to demonstrate that other approaches to the problem of image acquisition do exist. The manufacturing of high quality optics is relatively expensive when compared to the cost of conic mirrors which could also be designed to capture a cylindrical field of view, as discussed later.

2.3.4 Conic Mirrors

Conic mirrors have been used in a number of sensing applications where the desired field of view was larger than could be accommodated by a conventional camera. They have been used in mobile robots for obstacle detection and distance estimation [33, 34], and in pipeline inspection applications[23]. The basic principle is to use a reflective

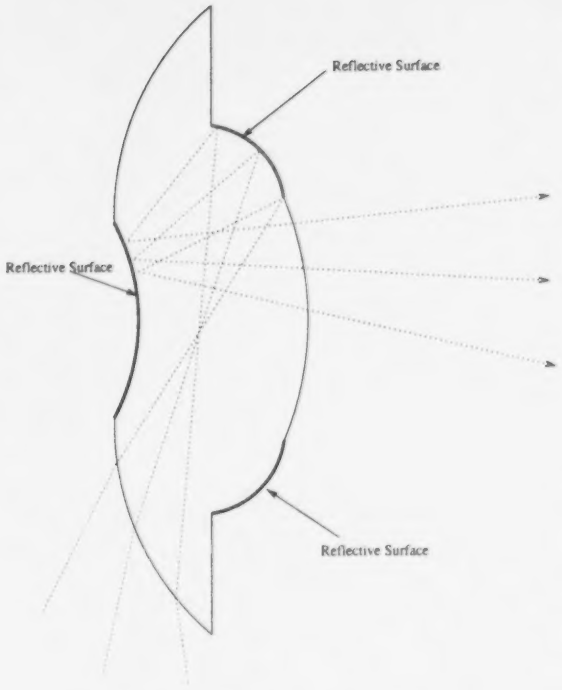


Figure 2.19: Panoramic Lens Block

surface that is generated by revolving the desired profile about an axis. The profile of this reflective surface can be changed to give different performance characteristics, such as increasing or decreasing the field of view, or distributing the resolution to different portions of the image. Figure 2.20 illustrates how a typical conic mirror could be used to capture a 360^0 field of view.

2.3.5 Panoramic Imaging Software

There are several panoramic imaging systems available commercially.

- **Microsoft Surround Video**

  The Microsoft Surround Video SDK[5] was released on July 17th, 1996. The basic functionality provided is a cylindrical transformation of panoramic images, and it is implemented as an Active X control. Microsoft's surround video requires the use of an expensive special panoramic panning camera. The camera has a slit that revolves around an imaging axis, exposing the film as the slit revolves.

[5] An overview of Surround Video is available at http://www.bdiamond.com/surround/overview.htm

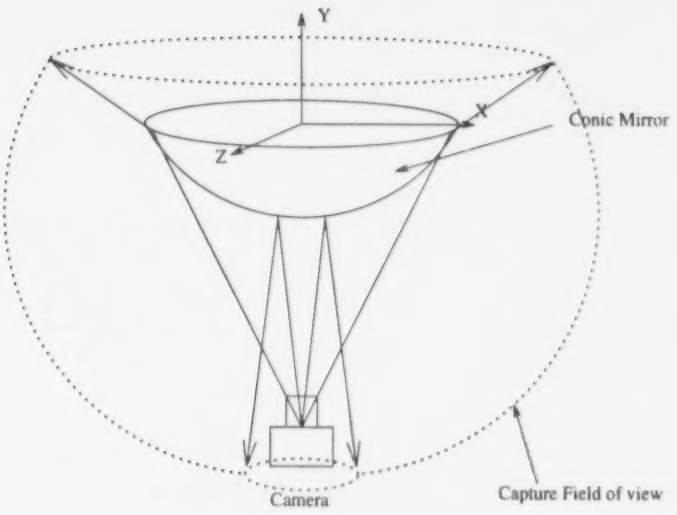


Figure 2.20: Conic Mirror Field of View

The photographer has to be careful to avoid being captured in the photograph by accident. The main disadvantage of this approach is the cost of the panoramic camera and the limited field of view which, like Apple's QuickTime VR, is only a cylindrical and not a spherical model.

- **Apple QuickTime VR**

  Apple's QuickTime VR system incorporates panoramic images into their QuickTime movies file format[10, 5]. The development kit includes software to stitch multiple conventional images to form a panoramic image. This stitching software can operate most of the time without the intervention of the user. However, for some scenes in which there are few distinguishing features, the software will not be able to find the corresponding features to do the stitching. In these cases user intervention is necessary. QuickTime VR offers a cylindrical panoramic viewing model.

- **Omniview**

  Omniview produces a product that will take two aligned fish-eye images and stitch them together to form a complete spherical view. This incorporates some automation in the preprocessing stage and involves the use of expensive fish-eye lenses. Often it requires manual intervention and touch ups using a paint

program [2]. The process of taking the two aligned photographs requires a skilled photographer. The major disadvantage of this method is that the fish-eye lenses are very expensive and must be made to order.

This type of immersive imaging technology allows multimedia content developers to present imagery of 3D scenes that would be either difficult, costly and/or impossible to do with other types of technologies. The obvious question is — why not use a video sequence to pan around a scene? A video sequence limits the degree of interactivity by allowing the user to go only backward or forward, effectively retracing the path that the camera took when the sequence was captured. Another disadvantage of video is the massive amounts of data required. Some of the problems can be addressed in a limited degree with the use of compression, such as MPEG or AVI codecs, or by using mosaic representation [13]. Another approach would be to capture an image of every conceivable viewing angle, in discrete increments, for a particular location. The images are then indexed such that as the user pans or tilts, the appropriate image is displayed. This is certainly possible, but would require massive amounts of storage as well as a special camera stand to allow for incremental pan and tilts between frames. That approach would also be costly to implement and use. The easiest and most economical method of panoramic image acquisition is to utilize the conic mirror, since it can be mounted on standard photographic and video cameras and does not involve stitching multiple images together.

2.4 Methodology

Software engineering is a relatively new discipline that tries to apply engineering principles to ensure a systematic approach to software development. This project incorporates some of the software engineering practices. There are many software engineering methodologies to choose from; some are better than others. We will examine a few of these methodologies and discuss some of their advantages and disadvantages. Based on these factors and the author's preference, we will pick a development methodology. It is not the author's intent to present the subject of software engineering, and proper treatment is beyond the scope of this thesis. We will merely give an overview of some of the key concepts. This will allow the informed reader to better understand the

process that was used to develop this software.

Object Oriented Analysis (OOA) and Object Oriented Design (OOD) is a relatively new way of designing and implementing software systems. The basic approach is fundamentally different from classical structured programming, which is more data centric. The object oriented approach solves problems in the problem domain rather than transforming the data to the solution domain. By this we mean that, in languages such as C or FORTRAN, the problem is analyzed and broken down into simple data types, and procedures or functions are designed and implemented to transform the data. These traditional techniques are based on the flow of data, which are often represented as data flow diagrams in the design phase. An object oriented approach, on the other hand, removes the separation between the data and the operations on the data. This has an important advantage — that of information hiding or encapsulation.

The object oriented approach was chosen because of its ability to encapsulate and modularize the software components. Since this is a research project, not all the requirements and problems are known in advance. Therefore, the ability to redesign and replace components in the implementation without impacting other portions is a desirable characteristic of a methodology. The overall methodology used was a combination of the spiral model, prototyping model, and object oriented methods.

The spiral model is basically a process that incrementally adds functionality to a product and/or incrementally modifies the project based on feedback from the customer. The process starts with the customer's initial requirements. Planning and analysis is done based on those requirements. A prototype is then designed and implemented. At this stage, with a prototype to show to the customer, other requirements are established, based on the customer's feedback. It is an evolutionary process until the customer has a product that meets his or her needs.

Prototyping is a process that is characterized by the emphasis on rapid development in order to demonstrate that the ideas can be implemented. In an ideal world, the prototypes should be thrown away, but this usually does not happen due to time constraints and limited resources allocated to the project. Prototypes are generally vehicles for learning and testing new ideas. They are usually not very robust and not intended to be industrial strength. One danger of this approach is that the customer

will get the mistaken impression that the prototype is the product, and that very little remains to be done to make the prototype into a product. The problem is that quality will be sacrificed if the prototype is retrofitted. It is the author's opinion that a partial solution to this problem is the use of object oriented techniques during the prototyping stage, as will be discussed later. Robust object oriented components can give rise to the re-usability that is desired, while still allowing rapid prototyping using these object oriented components.

The classical waterfall model of software development is, in the author's opinion, unrealistic for this particular project. The process of requirement specification, analysis, design, implementation, testing, and maintenance are more interdependent than the classical waterfall model would have us believe. In the author's opinion, design and implementation can be done concurrently. Often the design phase involves the use of other fancy notations that must later be translated into code during the implementation phase. The design phase can be done in the chosen Object Oriented language such as C++, and scenarios can be tested using the specified interfaces or methods. The interfaces can be reasonably complete and robust before the implementation begins.

Object oriented computing encompasses far more than just OOA and OOD; it has become mainstream computing. One example is the use of COM (Component Object Model) in the popular Windows 95 and Windows NT operating systems. Microsoft has indicated that COM will be the bedrock of the future versions of their operating systems[14]. Many subsystems in Windows 95 are in fact COM objects. In other words, these subsystems are systems level objects. The use of object oriented techniques is in line with the current trend in software development.

Chapter 3

Requirements

The objective is to produce an affordable package that is also easy to use for both the end user and the content author. It must be cost effective in terms of content creation and end user system requirements. We will specify the hardware requirements necessary to provide adequate performance for the software system, the functionality to be provided by the software, and the user interface requirements.

3.1 Recommended Minimal Target Platform

The target platform will be PC based[1]. Due to the open architecture of the PC, there are many possible configurations, making it difficult to specify a target platform. There have been efforts to define and use a set of standards that will guarantee a minimum level of performance. One of these standards is MPC (Multimedia PC) [9], which specifies the minimum requirements to qualify for one of three levels. Our target platform is MPC3 which includes a 75MHz Pentium (or equivalent) processor, 8 megabytes of RAM, quadspeed CD-ROM, etc. The MPC standard includes minimum requirements for both hardware and software. We will specify our system requirements as MPC3 hardware, and a 32bit operating system such as Windows 95 and Windows NT 3.51 (or later). These requirements are only a rough guideline. Some 486 based systems with a 3D graphics card may provide adequate performance. The software should be designed in such a way that it takes advantage of 3D graphics hardware, if present, but should also function in the absence of it. Since the performance of PCs doubles every 18 months, it is not critical if the software falls short

[1]PC refers to IBM PCs and clones or, more accurately a Microsoft Windows based system.

and requires a slightly faster system.

3.2 Viewing with a Virtual Camera

The desired system will have to take input images that are warped and do the appropriate transformations to correct the distortions and display the image as if it were viewed with a more conventional camera. The system must allow the user to look in any direction in an interactive manner, provided that the input image has that information. For instance, panoramic images can be viewed by panning the camera left or right and zooming in or out, but looking up or down would not be possible.

3.2.1 Panoramic Images

The term *panoramic* is used to describe images or pictures with a large field of view. Sometimes images with larger than 90° field of view are referred to as panoramic. We will use the term to mean truly panoramic or 360° field of view. Panoramic images are modeled as the surface of a cylinder. We can imagine a label on a can being the panoramic image. If we peel the label off and flatten it, we would have a panoramic image. These images will have a characteristic distortion due to perspective foreshortening.

Panoramic images can be acquired in many ways. There are panoramic scanning cameras that have a slit and, as the slit is revolved 360° about an optical axis, the film is exposed to the incoming light rays. Panoramic images can also be generated with computer graphics programs, such as ray tracing programs, that support panoramic cameras. Povray[2] is one such program. Other methods of obtaining panoramic images include taking several conventional images from the same spot, with the combined field of view that covers the full 360°. In this case, more processing is required to warp the images and combine it into a panoramic image. Software is used to stitch several images together to form a complete panoramic image. There are several problems with this approach, one being that different parts of the same scene can have different lighting conditions. When the pictures are stitched together, the seams may be noticeable. It may also be difficult to match up the seams exactly. This problem

[2]povray can be obtained at http://www.povray.org

can be caused at several stages: if the camera is misaligned when the pictures are taken; when the film is developed and made into prints; or when the film is scanned into the computer. Assuming a pinhole camera, the focal point must be in the same place for all the pictures or else there could be parallax[3]. The problem of image mosaicing for panoramic applications was addressed by Szeliski [25]. Another way of generating panoramic images is from the fish-eye or panospheric image formats.

Panoramic Texture Image Format

The texture format for panoramic images is a rectangular image, such as the one in Figure 3.1, which contains a full 360° field of view. Notice the panoramic distortions present in the image caused by perspective foreshortening. Due to perspective fore-

Figure 3.1: Panoramic Images

shortening, objects that are further away will project to a smaller area than objects closer to the camera. In Figure 3.2(a) the camera is located at the center of the cylin-

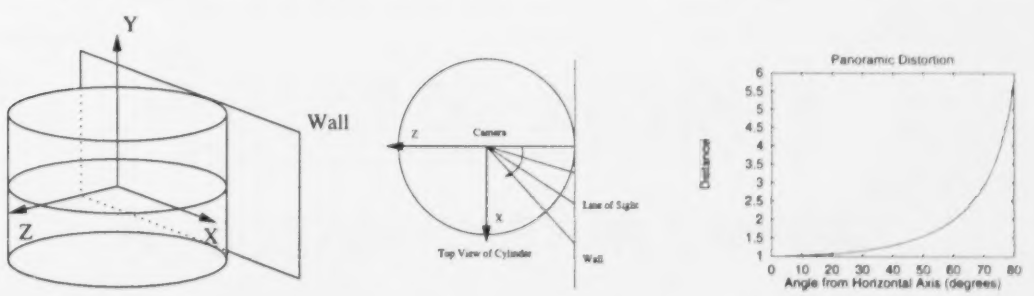


Figure 3.2: Panoramic Distortion (a) Top view of cylinder (b) Plot of distance to the wall

der, as seen from a top view. If we were looking at the wall in the diagram, we could

[3]Apparent displacement, or difference in the apparent position

plot the distance of the wall from the center of the cylinder as we look around in a clockwise direction (assuming that the cylinder has a unit radius). As we can see from Figure 3.2(b), the distance does not increase in a linear fashion. This implies that straight edges in the real world, that are not vertical when projected on the cylinder, would map to curved lines in the panoramic image. This is illustrated by the plot of the height of the wall as projected onto the cylindrical surface in Figure 3.3. It is

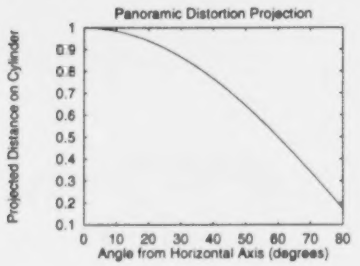


Figure 3.3: Height of Wall Projected onto a cylindrical surface

for this reason that viewing panoramic images with a smaller field of view, as in Figure 3.4, is not as simple as clipping the input image; we must correct the distortions.

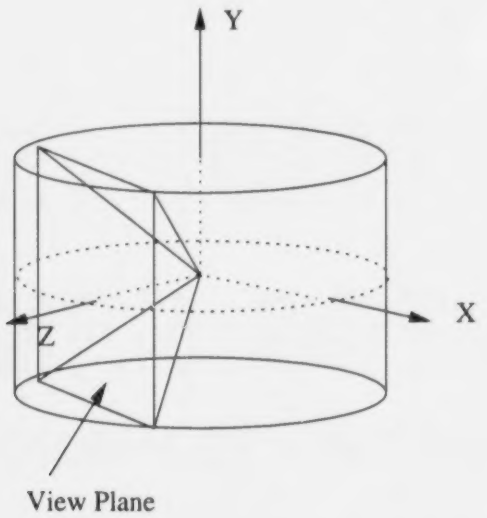


Figure 3.4: 3D Model of Panoramic Images

These distortions are more apparent in the panoramic images in Figure 3.5. The two images are shifted versions of each other, to demonstrate to the reader that each image is indeed a complete 360° around. Notice that the door in the lower image is

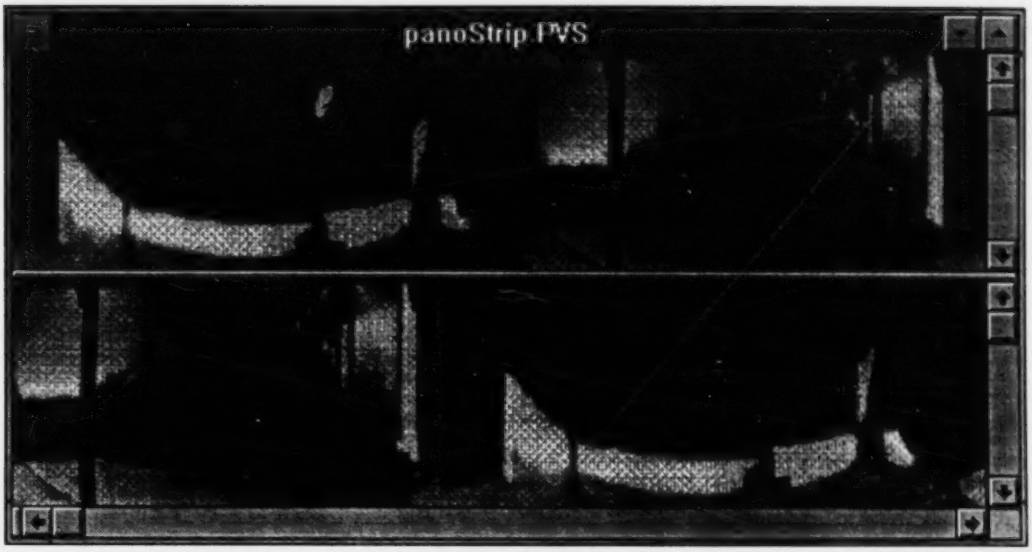


Figure 3.5: Panoramic Images

wrapped around in the upper image. Note also that the shadow and the wall have the characteristic panoramic distortion due to perspective foreshortening. The software is required to support pan and zoom operations with a virtual camera.

3.2.2 Fish-eye Images

Fish-eye images are taken with a fish-eye lens, which captures a hemisphere of the scene; usually the resolution is poor at the edges and better towards the middle of the image. Figure 3.6 is an example of a fish-eye image[4]. Fish-eye lenses are expensive and are not mass produced. The software will be required to support pan, tilt and zoom operations with a virtual camera.

3.2.3 Front and Back Fish-eye Hemispheric Image Pairs

The front and back fish-eye hemispheric image pairs are two calibrated fish-eye images taken with the camera pointing in opposite directions. Figure 3.7 illustrates the 3D model. The circular seam for the two halves is located in the XY plane. Figure 3.8 is a sample of an image pair taken with a fish-eye lens. These images are only roughly aligned since no special equipment was used when the photographs were taken. The

[4]Image courtesy of Steve Bogner at Piercorp

Figure 3.6: Fish-eye image of Ottawa

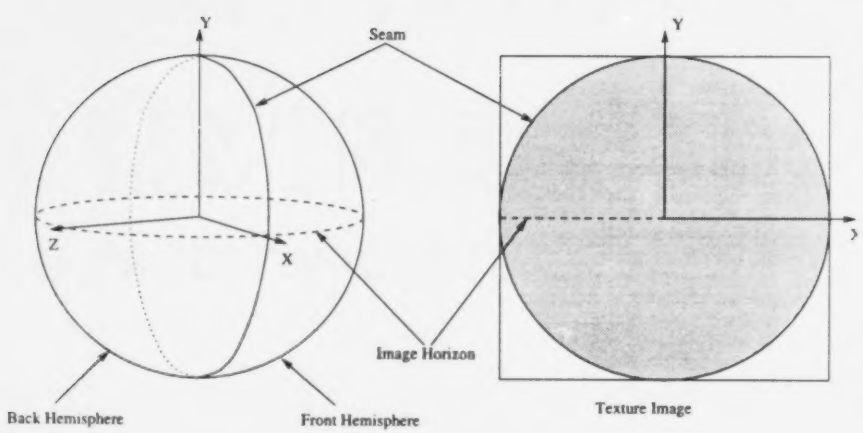


Figure 3.7: Front and Back Hemispheric Model

photographs were scanned into a computer, then cropped (clipped) and resampled. The software will be required to provide a distortion-free view from a virtual camera

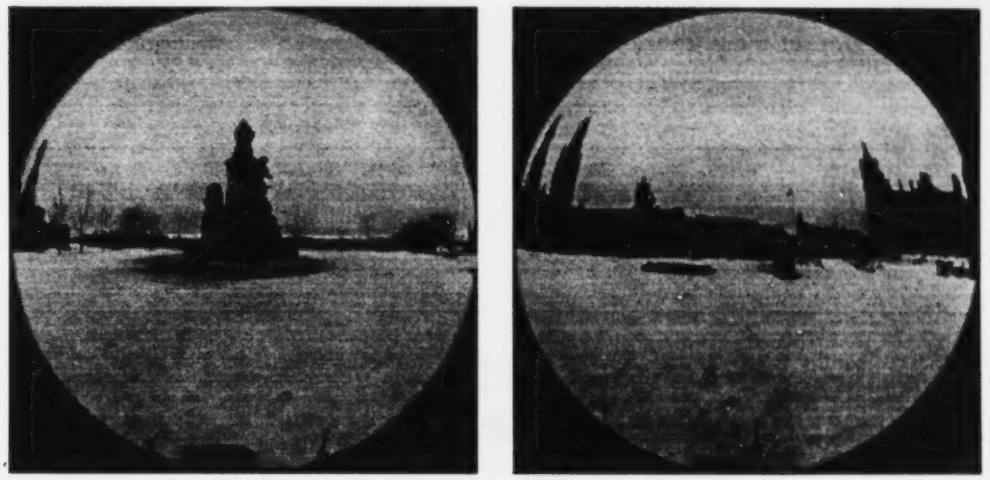


Figure 3.8: Front and Back Fish-eye Panospheric Image Pair (Ottawa)

supporting the pan, tilt, and zoom operations.

3.2.4 Upper and Lower Hemispheric Image pairs

The upper and lower hemispheric image pairs are illustrated in Figure 3.9. As we can see from the 3D model, the seam for the two halves is at the image horizon. We could map other formats (such as panospheric) to this format in a preprocessing step. The greatest advantage of this format is that the distribution of the texels is better.

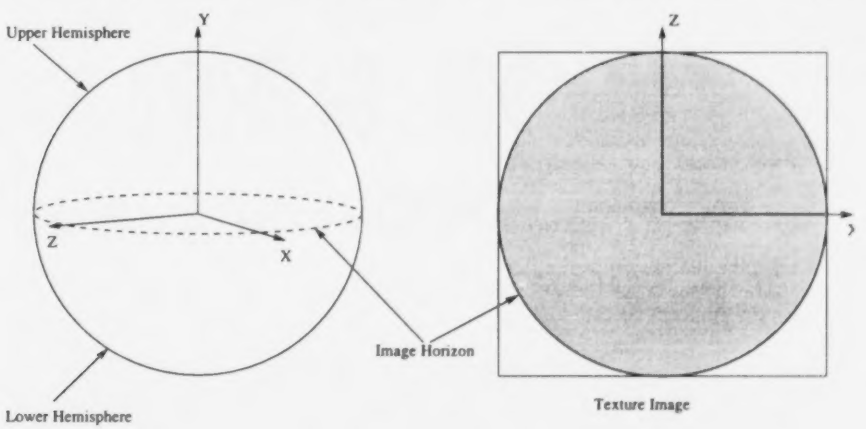


Figure 3.9: Upper and Lower Hemispheric Image pairs

3.2.5 Panospheric Images

The panospheric optic being designed by PVSI is shown in Figure 3.10. This optic will provide larger than 70% field of view.

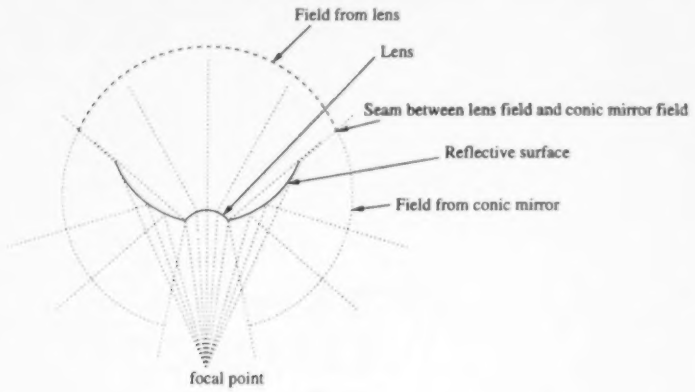


Figure 3.10: PVSI's Panospheric Optic

The 3D model for panospheric images is shown in Figure 3.11. Notice that the panoramic field from **a** to **c** is captured with the reflective conic section, and that the seam between the lens field and the reflect field is not continuous in the captured image.

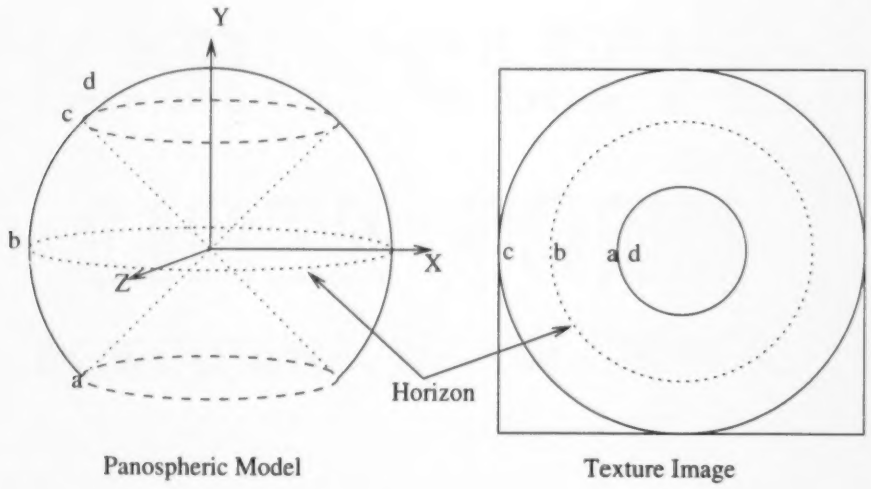


Figure 3.11: 3D Panospheric Model and the corresponding image plane

A sample input image is shown in Figure 3.12

Figure 3.12: Sample Panospheric Image

Panoramic Images from Panospheric Images

It is both possible and useful to map panospheric images to the panoramic format. This particular transform is interesting in that it allows the viewer to see a full 360° in a rectangular area. The resultant image will be of the format that can be used in the cylindrical panoramic model discussed previously.

3.3 Image Acquisition

Our proposed system will use the panospheric optic, being developed by PVSI, to capture images in the panospheric format described previously (section 3.2.5). The major advantage is that preprocessing will be reduced dramatically, and the overall system will be much more economical to produce and easier to use.

3.4 User Interface

The user interface (UI) requirements must conform to standard user interfaces as much as possible, as it takes users far less time to learn a standard interface than it does a custom interface. These requirements include the look-and-feel and the functionality. Microsoft has invested considerable resources into usability studies for the Windows 95 operating system, and has supplied a set of guidelines for the design of user interfaces for Windows 95/NT operating systems[16].

The key concepts are summarized below.

- The program should have a user centered design, that is to say that the user should be in control and should feel in control. One implication of this is that the user, rather than the program should initiate all the actions. Graphical User Interfaces (GUIs) satisfy this requirement since they are event driven; the program simply responds to events generated. Another implication is that the program should be as interactive and responsive as possible.

- Familiar metaphors provide a direct and intuitive interface to user tasks. By allowing users to transfer their knowledge and experience, metaphors make it easier to predict and learn the behaviors of software-based representations. When using metaphors, we need not limit a computer-based implementation to its real world counterpart. Metaphors support user recognition rather than recollection. In this particular application, the natural and obvious choice is to use the camera metaphor.

- The user interface should be consistent within the product, with the operating environment and with the metaphor. In our application, product consistency would involve features such as being able to pan or tilt the camera in the same manner no matter what the underlying image transform was. For instance, clicking on the left region of the view window should pan the camera to the left, regardless of whether it is a panoramic or panospheric image. Consistency with the operating environment would mean features such as keeping the look and feel of the application as similar to the standard applications as possible. This includes, for instance, using the standard dialog boxes where appropriate

instead of building a custom one.

- The user interface should be forgiving when the user makes errors. Many users learn to use new programs by trial and error; therefore, it is prudent to warn users of situations where damage to data could occur. The user interface should allow the user to recover from errors and always return to a known state if the last operation could not be completed. For instance, if a user is attempting to save a file but the disk is full, the program should notify the user of the problem and abort the operation instead of terminating the whole program.

- The program should give appropriate and timely feedback. This could include changing the cursor to reflect the type of operations that the user can perform, as it is moved over certain objects on the screen. For lengthy tasks, an appropriate progress indicator should be used. Since we are using the regions on the screen to dictate the direction that the camera moves, appropriate cursors should be shown as the mouse moves in the window.

- The user interface should have a balance between functionality and simplicity. Progressive disclosure property-sheets are a good way of progressively disclosing the parameters to the user while not overwhelming them with too many settings at any one time. Property-sheets provide a way of categorizing settings in the form of tabbed pages. Each tabbed page is the equivalent of a dialog box. Using this method, many of the parameters can be exposed to the user, but only relevant settings are shown for a particular task the user wishes to initiate.

- Users can be categorized into two groups: beginner and advanced, who traditionally show different patterns of usage. The UI should accommodate both groups. Advanced users want efficiency, and there are many things that can be done to give them this efficiency without introducing complexity to the interface. For instance, accelerator keys can be defined so that advanced users can utilize keyboard shortcuts to perform certain operations. Many advanced users find this more convenient because their hand never has to leave the keyboard to reach for the mouse.

- The use of pop-up menus can emphasize the object oriented nature of the software; they provide a context sensitive way to access the functionality and properties of the selected object. Context sensitive pop-up menus are used extensively in Windows 95 and fit into the object oriented metaphor. The menu options correspond to the methods exposed by the objects.

- Statusbars and toolbars are common in most windows applications. Statusbars can be used to give users feedback, while toolbars can be used to provide shortcuts to commands.

The software should be designed to incorporate as many of these general guidelines as possible. In addition, the UI must facilitate the creation of multiple views of the same scene.

3.5 Technical and Economic Feasibility

A comprehensive feasibility analysis is beyond the scope of this thesis. However there are a few factors that would seem to suggest that this project is both economically and technically feasible.

Since this is an emerging market, it is hard to accurately predict the demand for such a technology. Informally, it is estimated that the vertical market will be large enough to justify the commercialization of a panospheric imaging system[1, 2, 21, 22] with all the potential applications mentioned elsewhere in this thesis.

The systems that do exist are either difficult to use and/or require expensive equipment or a skilled operator in the authoring process [2]. The authoring systems that require a skilled operator will ultimately cost more because the content creators' time is valuable. This proposed system will require far less manual intervention and does not require a skilled operator to create the content. Figure 3.13 shows roughly, the cost of various panoramic image acquisition technologies.

System	Cost (US)	Availability
Panospheric (Conic Mirror) Optic	<$1,000	Can be mass produced
Revolving Panoramic Camera	$2,500	
Fish-eye Lens	$25,000	Very limited quantity — must be custom ordered

Figure 3.13: Cost of Various Panoramic Image Acquisition Technologies

Chapter 4

Design

4.1 Choice of Platforms

Design and implementation decisions are often dictated by the system requirements and objectives, and this project is no different. Our predominant objective is the eventual commercialization of the immersive imaging technology. As I will discuss later, this has influenced many of the design and implementation decisions.

4.1.1 Operating System

Given the time constraints, it was decided early on that supporting multiple operating systems would not be feasible. The need to reach mass markets forces us to target the most widely used operating system. The clear winner by that metric is, without a doubt, Microsoft Windows 95. Windows 95 was originally intended as a transition Operating System (OS) to the much more robust Windows NT. Windows 95 is not a subset of Windows NT; on the contrary, Windows 95 actually has much better multimedia subsystems and supports far more devices. There are features that are exclusive to each of the two operating systems. With some careful planning, the software can be implemented in such a way as to be compatible with both Windows NT and Windows 95. At the heart of both Windows NT and Windows 95 is the 32 bit Windows Application Programming Interface (Win32 API) [15]. It is necessary to restrict ourselves to the set of common API calls if we are to target both operating systems. Most of these concerns need not be addressed by the programmer explicitly if the proper tools are chosen.

4.1.2 Programming Language

Using an object oriented approach does not restrict us to object oriented languages such as C++, Java, or Smalltalk. It is possible to implement object oriented software in non object oriented languages, such as C or Pascal, but it is not as convenient. The adage that it is possible to write good programs in any language and it is also possible to write bad programs in any language certainly applies.

The choice of programming language was clear, given that the target OS was Windows 95/NT. At this point in time, C++ has the most robust support tools — other than perhaps Microsoft Visual Basic (VB). While Visual Basic is good for rapid prototyping and building GUIs (Graphical User Interfaces), the calculation intensive parts would probably be too slow. In the author's opinion, the major advantage of C++ over Visual Basic in this particular project, is the object oriented features of C++, and not so much the speed considerations. With the VB approach, CPU intensive operations can be delegated and programmed in other languages such as C/C++, if necessary. Object oriented programs are easier to develop and maintain in C++.

Another object oriented language to consider is Java, Sun Microsystems' object oriented programming language. Java is essentially a cleaned up version of C++ where the arguably bad features were eliminated. Multiple inheritance, the ability to forge pointers, operator overloading, and implementation dependent features are not present in Java, as they are in C++. Instead, Java offers garbage collection, which eliminates the need for pointers; and Interfaces which provide the functionality of multiple inheritance[24]. While Java offers many other benefits over C++, it was still in beta state at the start of this project and there were very few development tools available; therefore, it was not seriously considered at that time. As will be discussed in Section 7.3.5, Java opens up many possibilities for Internet Applications using this imaging technology.

There are several popular C++ compilers available for Windows development. Because of the market dominance of Microsoft Visual C++, and the reasonable quality of the compiler in terms of conformance with the latest C++ language standards, it is a reasonable choice. At first glance, it may seem that the discussion of which

programming language or compiler to use is a waste of time, but nothing could be further from the truth. The statement "If all you have is a hammer, you treat everything like a nail." rings true. The choice of development tools has a major impact on the design and implementation of software. We will be designing our software architecture around the application framework provided in the form of the Microsoft Foundation Classes (MFC) with the Visual C++ development tool.

4.2 Design Considerations

In object oriented design it is often necessary to take into consideration the features, strengths, and weaknesses of a specific language. The design process that was used focuses on identifying objects and defining interfaces or methods[1]. Sometimes this is done on cards, each one representing an object with its interface listed on the cards. In the author's opinion, this is not necessary; it can be done using an actual Object Oriented language such as C++. The interface can then be designed that will suit the implementation language. After these interfaces are defined, and are robust enough to pass the criterion of being reusable, scenarios of communication between the objects can be examined and tested to ensure the interfaces for the objects are adequate. After scenarios are exercised successfully, the implementations of the methods can begin. All object oriented languages support inheritance. It is a common way to promote code reuse in object oriented designs and will be used where appropriate. Dynamic binding, on the other hand, has a runtime cost; therefore, its use was limited as much as possible to infrequently used classes. A much more efficient mechanism is to use the C++ template feature to define interfaces, following the principles in the Standard Template Library (STL).

4.2.1 Software Re-usability and Performance

One of the major challenges facing software engineers is software re-usability. In the author's experience, an object oriented approach is a step in the right direction. Recent additions to the C++ language, such as the Standard Template Library (STL),

[1] In the object oriented paradigm, interfaces and methods both refer to the operations that can be performed on the object, and will be used interchangeably

is an important contribution toward the goal of re-usability. Some image processing filters can be implemented using templates. This would essentially factor out all the supporting code such as the iterators — something that is not possible to do in a *type-safe* manner in some languages. In C, if we factored down to such a fine granularity, we would incur the cost of a function call on every operation. In C++, however, we can instantiate an object with an inlined function operator. This basically gives us the fine granularity in a type-safe manner, while removing the cost of the function call, because the inlined functions have compile time complexity. With some knowledge of the features in the implementation language, we can design in a manner that promotes software reuse while not sacrificing performance.

4.2.2 User Interface

To design and implement a user interface that conforms to the requirements is a major undertaking. However, our task is made much easier by application frameworks that are commercially available. Application frameworks are, basically, a collection of C++ classes that encapsulate many of the common objects required in the development of most applications. Simply put, they are ready made generic components that can be used and modified to do application specific tasks. Application frameworks provide the infrastructure for most types of programs; they are like empty new houses, ready to be furnished and decorated. The major advantage of using an application framework such as Microsoft Foundation Classes (MFC), is precisely for the built-in level of conformity to user interface guidelines. We are not saying that design of the user interface is not required, but that the design will be easier to implement with the help of MFC.

4.3 Software Architecture

The software architecture will follow standard practices in use today by Microsoft Windows software developers. It implements a Multiple Document Interface (MDI), where the main window of the application can contain more than one document and each document can contain more than one view window. Figure 4.1 illustrates the general (simplified) relationship between the application, documents, and the

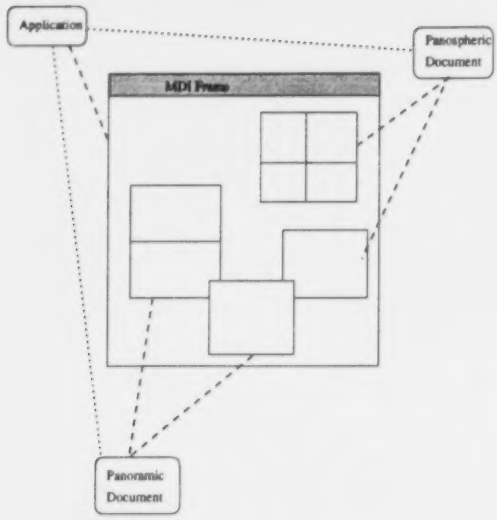


Figure 4.1: Multiple Document Interface

views. Figure 4.2 is an actual screen shot of the final software product that roughly corresponds to Figure 4.1. The application object is involved with the management and creation of the documents, and the main MDI frame window. The MDI Frame window manages all the MDI child windows, such as all the views associated with the documents. There are two hierarchies involved here. One is for the non visual objects such as the application object, the document template object(s), and the document object(s). The other hierarchy is the visual objects hierarchy, which includes all the windows that are seen on the screen. The visual objects are actually Windows objects wrapped by a parallel C++ object hierarchy. Strictly speaking, the windows hierarchy does not require the existence of the parallel C++ hierarchy. Once the MDI frame is created, the application can now receive messages or process events generated by the user. These events could include selecting menu options, moving the mouse, clicking on the mouse buttons, etc. At this point, the user is in control and can initiate events. How these events are handled will be covered later when we discuss message dispatching. But first we will take a closer look at the Document/View model.

4.3.1 Document/View Model

The software is designed around the document/view model, which is used to encapsulate the data in the document and provide one or more views into the document.

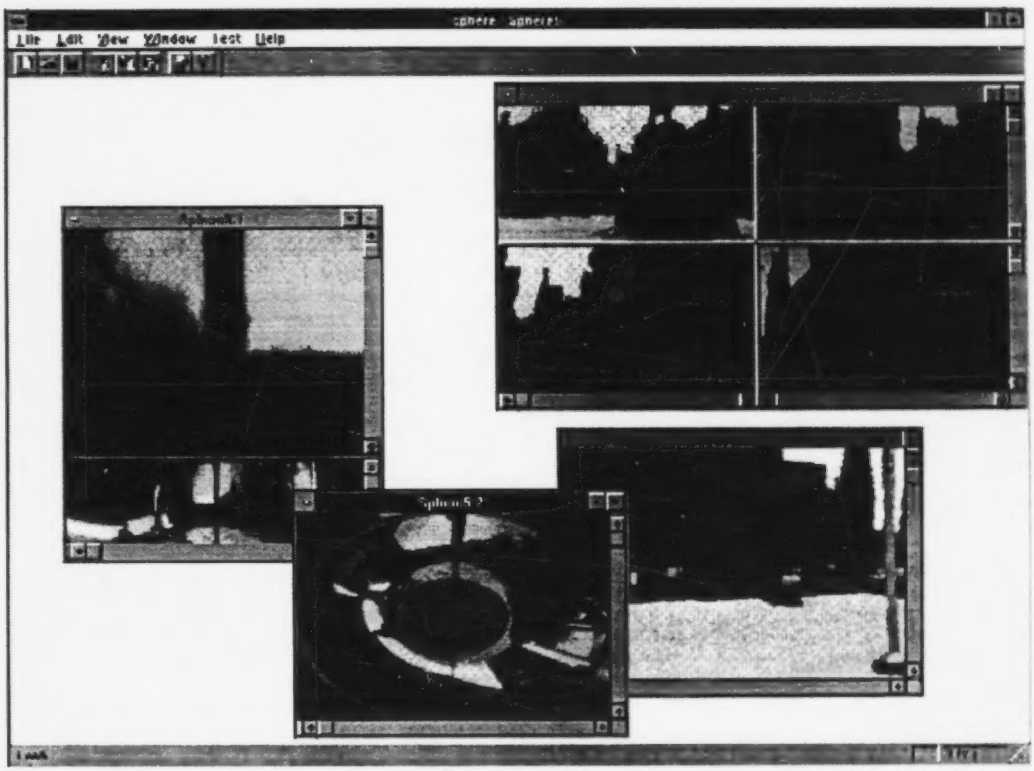


Figure 4.2: Multiple Document Interface Screen Shot

This is a generic object oriented approach that is commonly used to decouple the data from the way that it is presented to the user. For this particular project, the document contains the warped images and the parameters that describe how that image was warped. One of the view types will be our virtual camera. In this way, we can have multiple cameras looking at a single scene, providing the user with the ability to manipulate the virtual cameras independently. We can use the notion of different view types to view the same image in different ways, using different transforms. Essentially, the document/view model allows the same data to be presented to the user in different ways, and will be used extensively in the software design. Figure 4.3 shows the basic relationships involved in maintaining a document/view model. The document template object is used to manage the creation of the document and the views; it keeps track of the document class and the associated view classes. When a new document is created, the document template object knows which document type to create and it instantiates a document. At this point the document contains default data, but is still not visible to the user because there is no view associated with it. The document template object now creates an appropriate view for the document object and informs the document object. The document object attaches the view to itself and, in this process, the view object also sets a back link to the document object. The document object maintains a list of view objects. This list allows the document to attach more than one view of the same or different type. The views are the main method through which the user manipulates the document or the data. If the user changes the document data through a view object, the document object informs all the other views to update their representations accordingly.

4.3.2 Message Dispatching

In graphical user interfaces, users interact with the programs by generating events. Events are generated for every action that the user takes, including moving the mouse, typing on the keyboard, clicking on a mouse button, etc. Message dispatching is a mechanism that is commonly used in graphical user interfaces to handle events. It is central to graphical user interfaces, and usually entails decoding the messages or events and calling the appropriate handler method. In simple applications this is not a major issue; one needs only associate the appropriate handler for the event of

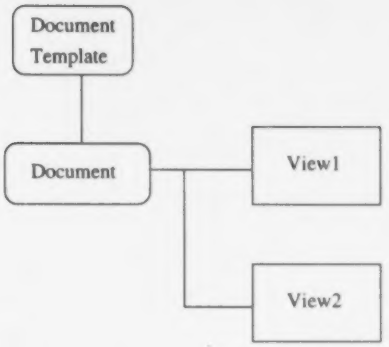


Figure 4.3: Document View Model

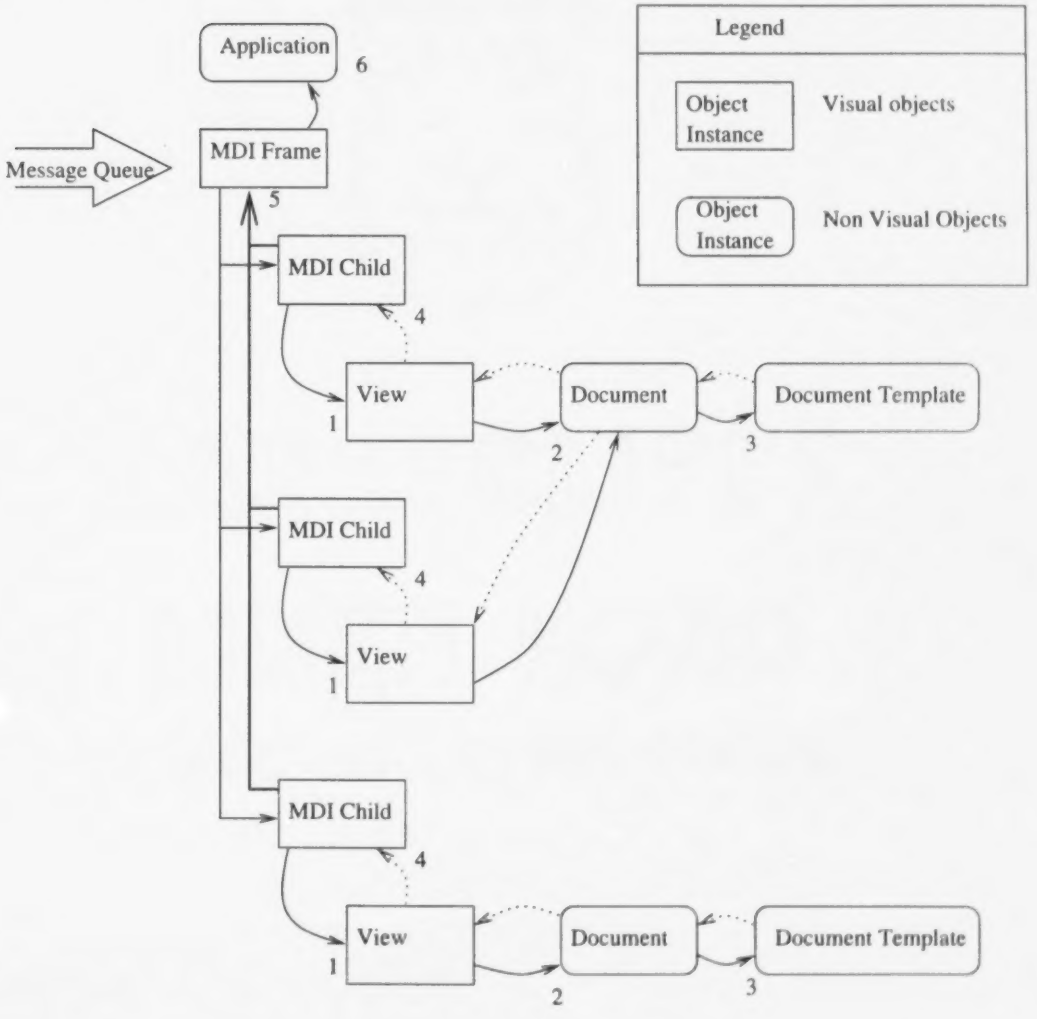


Figure 4.4: MFC Message Dispatch Mechanism

interest. In the context of the MFC application framework, messages are dispatched in the manner depicted in Figure 4.4. Under 32 bit Windows, each application has its own message queue. Events are generated when the user interacts with the application using the various input devices, such as the mouse or keyboard. When the application receives its time slice and is executing, it checks its message queue. The message is first passed to the MDI Frame which then passes it to the active Child MDI window. (Recall that the MDI Frame can contain many MDI Child windows.) The active MDI Child window then passes the message to its view window. The view gets the first chance to handle the message. If the view doesn't handle that message, it is passed to the view's document object. If the document object doesn't handle the message, it is passed to the document template object. If it is still not handled, the message propagates back up the hierarchy to the MDI Child window. If the MDI Child window doesn't handle the message, it is passed back to the MDI Frame window. If at this point the message is still not handled, it is passed to the application object for a last resort attempt to handle it. All the standard messages have a default handler somewhere along this search path. Similarly, all our custom commands and messages should have a handler installed somewhere along this search path — ideally at the place where it makes the most logical sense.

The important idea here is that the message handling can be overridden at any point in the search process. The implication for design is that the messages should be handled at appropriate places, and not intercepted; they should be classified such that the handler can be placed in the appropriate location. For instance, a view object should not implement a message handler for messages intended for the document object.

4.4 File and Texture Image Format

There are two stages at which the texture image format should be considered: first, when it is stored in persistent storage such as a hard-drive or CD-ROM; second, when it is stored in main memory for transformation. When the images are stored on disk or transferred across a network we want them to take up as little space as possible. To this end, we will use a standard image format such as JPEG or GIF to compress and

store the images. Our file format will include links to these texture files, as shown in Figure 4.5. The advantage of using links is that the texture file can be viewed

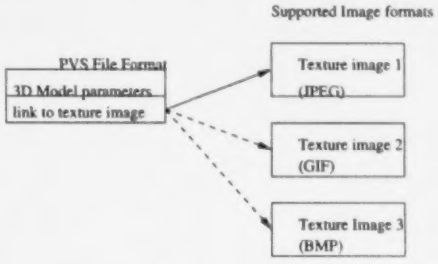


Figure 4.5: PVS file format

and processed by any program that supports those standard image formats. The disadvantage is that if the user moves those image files, the links will be broken. The texture images must be preprocessed or resampled to dimensions that are a power of 2. This restriction will facilitate much faster texture coordinate calculations, since bitwise shift operations can be used instead of multiplications. In practice this is not a severe restriction, because the input images can be scanned in at much higher resolutions than can be handled as texture maps. Since resampling is a preprocessing step, we can apply more expensive filtering techniques.

The format of the content of these texture images also needs to be considered. The circular texture images are kept in that format because the area of the resultant texture image is less than that of a transformed image. If we were to preprocess the circular texture images and transform them into rectangular representations, there would be redundant data in the rectangular representation that is not present in the circular representation. In this transform, the pixels closer to the center would be stretched more, and the texture mapping process becomes slightly more complicated to calculate. When the images are loaded from files, they are decompressed to a buffer in main memory for texture mapping, as shown in Figure 4.6. The texture images in main memory cannot be compressed since we need fast random access to the texels during texture mapping.

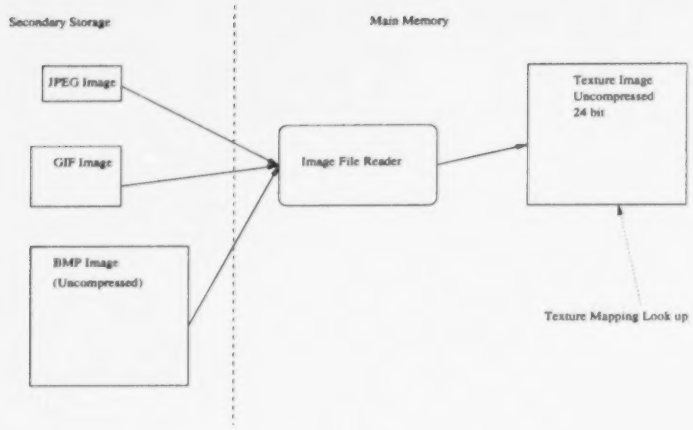


Figure 4.6: Image Storage format

4.5 Object Space to Texture Space Mapping

Texture mapping is central to this project. It is the method by which the flat 2D input images are mapped to 3D objects. The 3D objects are in turn transformed into a perspective correct view as seen by a virtual camera.

Generally, all the transforms are computed in the following manner:

1. determine the virtual camera parameters.
2. determine the appropriate 3D surface onto which to map the images (based on the selected 3D model).
3. determine the texture coordinates corresponding to the 3D surface points.
4. render the surface.

Virtual Camera Parameters

The virtual camera parameters include the vertical and horizontal field of view, specified in degrees or radians. These parameters are affected by the zoom operation, but not by the pan and tilt operations. Zooming in can be modeled as decreasing the field of view. Conversely, zooming out is modeled as increasing the field of view. In order to preserve the aspect ratio for the virtual camera, the vertical and horizontal fields of view are both multiplied by a scaling factor. Figure 4.7 shows the virtual camera

located at the origin. We can see that by increasing the field of view, a larger surface area is visible to the camera. Because the camera's view is then mapped to a viewing window, which is of a constant size during the zoom operation, it is equivalent to zooming out. In other words, more of the scene will be visible, therefore the objects in the scene will be smaller. Each virtual camera view also contain parameters for the

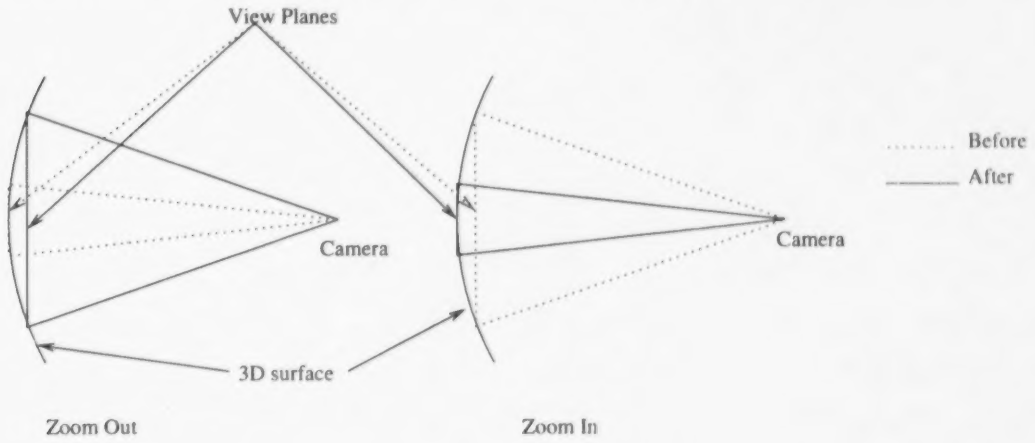


Figure 4.7: Zooming using a virtual camera

ground plane angle and elevation angle. The camera direction vector can be specified in terms of two angles, as illustrated in Figure 4.8. The ground plane angle is the

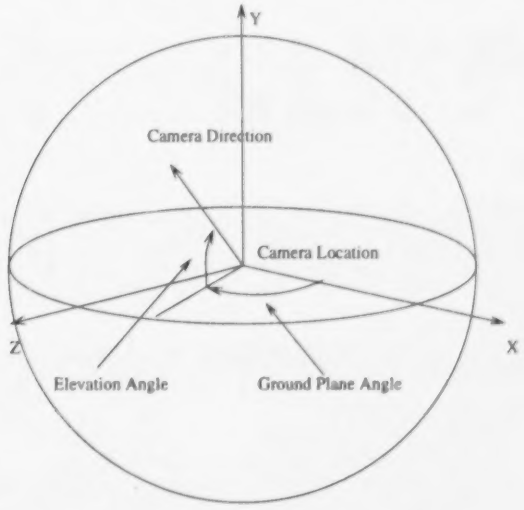


Figure 4.8: Camera Direction

rotation angle about the Y axis, and the elevation angle is the smallest angle between

the camera direction vector and the XZ plane, or the ground plane. Essentially this is similar to the spherical coordinate used in mathematics, where θ is the ground plane angle and ϕ is the elevation angle. The difference is that the elevation angle is defined to be zero when the camera direction vector is on the XZ or ground plane, and ranges from $+90°$ to $-90°$, (whereas ϕ ranges from $0°$ to $180°$), when the camera vector is straight up and straight down, respectively. The camera direction vector actually represents the direction of the center of the camera. By changing the ground plane angle one would be effectively panning the camera. Similarly, by changing the elevation angle one could achieve tilting, or in effect make the camera look up or down. The observant reader can conclude that if we change the elevation angle to a value outside the range of $+90°$ to $-90°$, we could look at the world upside down. It was decided that the ability to look at the world upside down would cause the casual user confusion, while offering very little benefit; therefore, the ranges defined above serve as the floor and ceiling values for these angles. That is to say, when the user points the camera straight up, that is as far as it will go; they cannot bend over backwards to look at the world upside down.

The user can access these camera parameters by using the mouse to move to the appropriate region in the window, as indicated in Figure 4.9. The cursor will update accordingly to indicate the direction that the camera would pan or tilt. The user can press and hold the left mouse button to perform the actual pan or tilt in the desired direction. If the mouse button is held down, the camera will continue to pan or tilt. The zoom operations are performed by the users moving to the + and - regions and clicking on the left mouse button. More advanced users can use the right mouse button to bring up a context sensitive menu for the view, and select the *property* option which will bring up a property sheet with the properties of the camera in one of the tabbed sheets, as shown in Figure 4.10. As we can see, the horizontal and vertical field of view angles can be adjusted independently from here. We can also perform pan and tilt operations at this point by adjusting the *Ground Plane Angle* and the *Elevation Angle*, respectively.

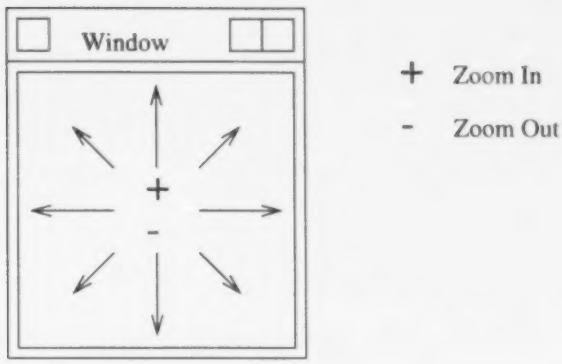


Figure 4.9: Active Window regions for Camera movement

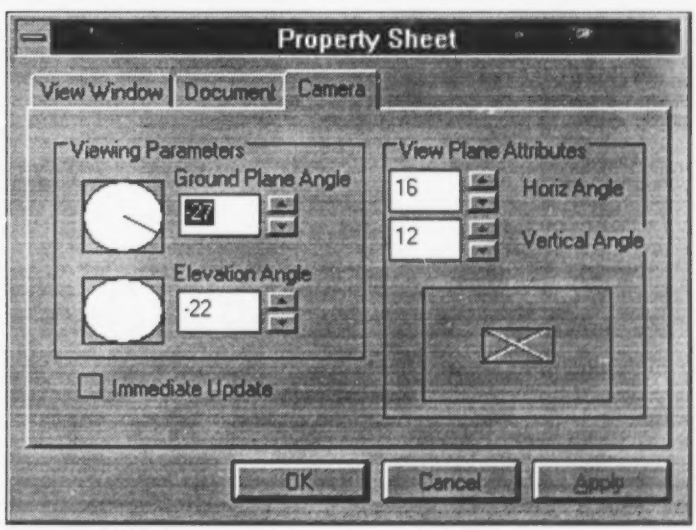


Figure 4.10: Property sheet for Camera Parameters

Determining the 3D surface

The surfaces involved can be modeled as quadratic patches. In order to specify the quadratic patches, we need the nine control points on the patch. This can be done very simply since we know the 3D object we are approximating with the quadratic patch, and the virtual camera's viewing parameters. To determine the control point for a spherical model we would transform the nine control points, as depicted in Figure 4.11, from viewport coordinates which are normalized to a range between

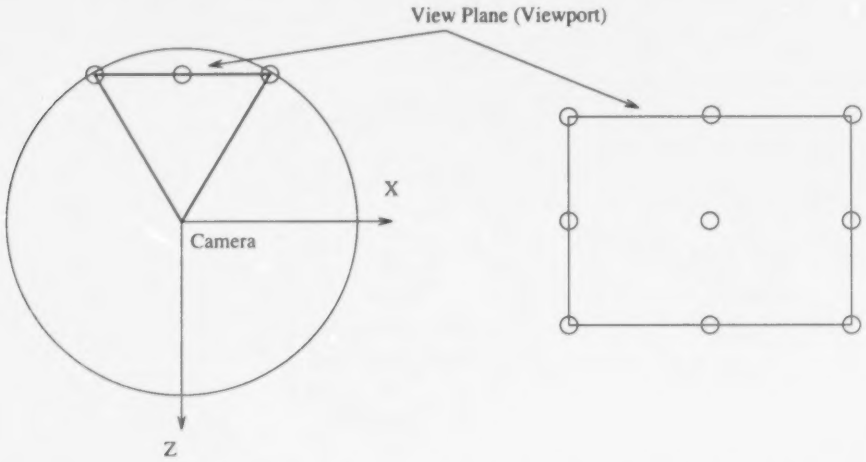


Figure 4.11: QPatch Control Points in Viewport Coordinates

0 and 1. The viewport coordinates are then transformed into camera coordinates. The camera coordinates are transformed to world coordinates. From there, the world coordinates are transformed into object coordinates. Because of our simplification, by locating the camera and object coordinates at the origin of the world coordinates, we need not perform some of the transforms as outlined above. The nine control points can be treated as vectors originating from the origin. As we can see in Figure 4.12, we can simply use these vectors to calculate the intersections with the object model, be it cylindrical or spherical. Once we have the intersection points, we have the control points for a quadratic patch[2] that approximates the 3D model.

Quadratic patches can be used only when we are modeling a conventional camera or planar geometric projections. If we want to produce views such as the panoramic

[2]fast rendering of quadratic patches is supported by most 3D graphics hardware.

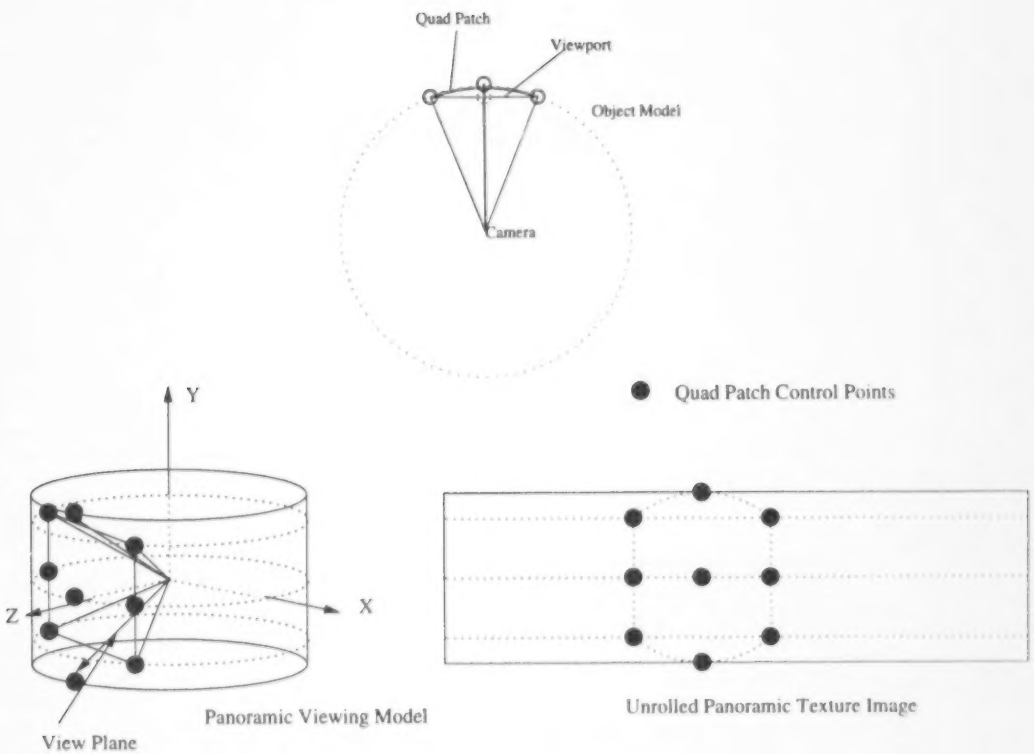


Figure 4.12: QPatch Control Points in Object Model Coordinates

strips, we must use a different approach since these are non planar projections. Figure 4.13 illustrates the general approach to the problem of viewing non planar projections. The 3D or conceptual model for a panoramic strip is that of a cylinder. In

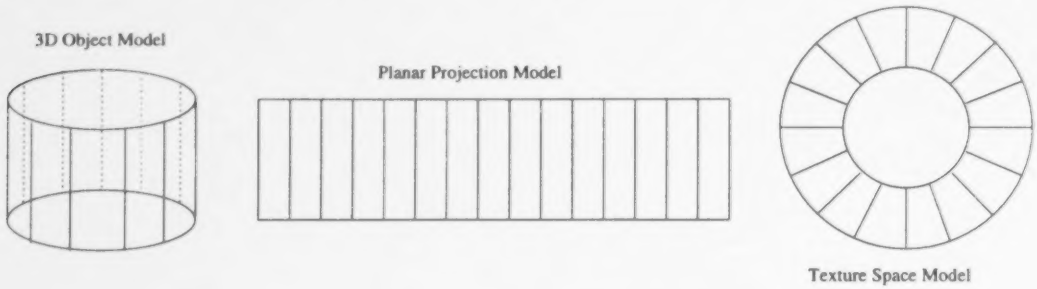


Figure 4.13: Panoramic Strip Viewing Model

order to view a complete 360° field of view, we *unroll* the cylinder much like peeling the label off a can. Now that we have a planar surface we can map it to a viewport which can then be seen in a window. The only problem that remains is to find the corresponding texture coordinates. If we are dealing with panospheric type projections then the corresponding texture space model is as illustrated in Figure 4.13.

Determining the Texture (Source Image) coordinates

Once we have the control points in object space, we need to determine the corresponding control points in texture coordinates.

For panoramic images the x direction corresponds to 360° about the Y axis in the 3D model. The y direction in the panoramic image corresponds to the vertical field of view of the camera. From this information we can map a point on the 3D model to a corresponding point in the texture image.

For the spherical models, the general idea is to map the 3D point by projecting it onto the texture plane. The panospheric model, however, involves the use of spherical coordinates. The radius of the 3D sphere is assumed to be 1. Any point on that sphere can be converted from Cartesian (x, y, z) to spherical (ρ, θ, ϕ) coordinates. The spherical coordinates can then be used as polar coordinates in the texture image, where ϕ is used as the r in the polar coordinates (r, θ). The general idea is illustrated in Figure 4.14. In reality, ϕ must be recalculated to reflect the fact that the field from the conic mirror causes a discontinuity in the texture image, as described previously.

The Polar coordinates can then be converted to Cartesian coordinates to correspond

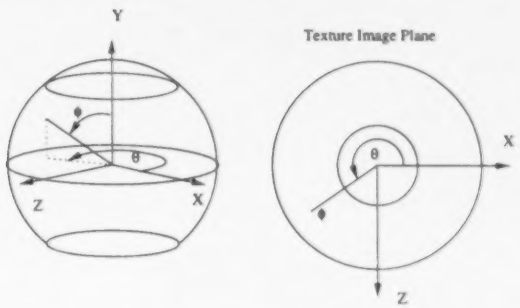


Figure 4.14: General principle in Panospheric Mapping

to the texture space coordinate system.

Distributing available resolution in spherical models

When mapping the image to a 3D surface there is the problem of determining how to distribute the available pixels. If we model the texture image as a projection of a perfect hemisphere, then there will be far fewer pixels at the edge of the active image region. One way to compensate for this is to model the texture as a projection of a smaller section of a sphere, as shown in Figure 4.15. The other way is to scale the sphere in one dimension so that it becomes an ellipsoid. In effect this allows us

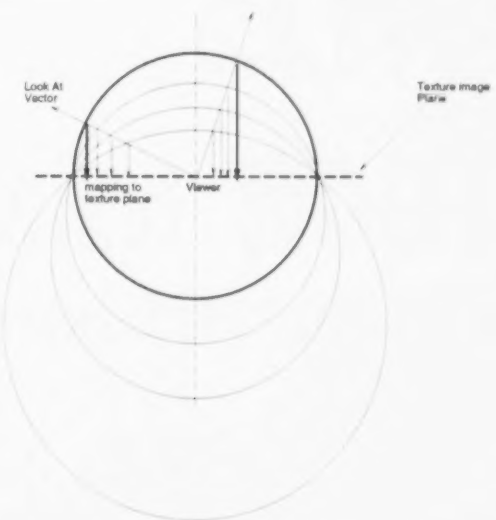


Figure 4.15: Resolution Distribution Model

to adjust the model such that it approximates the model of the imaging sensor that was used to capture the texture image. This is useful for images for which the model parameters are not known or difficult to determine.

Rendering

Now that we have two sets of control points (one set for the 3D surface and the other set for the texture space), we can iterate over the u,v parametric space to find the texel that corresponds to a given location on the surface. In effect, we are mapping the texels onto the surface. To explain this in terms of sampling, we are reconstructing the original scene with our 3D surface model. We are not finished yet, because we must now resample the scene as viewed with a perspective virtual camera. It is the perspective transform of the virtual camera model when applied to the 3D model of the scene that actually corrects the distortions.

Every time the view changes because the user interacts with the program, the object space to texture space mapping is performed and the view is updated.

4.6 Integration with Other Applications

The key to the success of most software applications in the Windows environment in the future will be the ability to integrate and interoperate with other applications as seamlessly as possible[14]. The key technology that will enable easier integration of software components is systems level object oriented technologies such as Component Object Model (COM), and derivative technologies such as OLE[3] and Active X. Microsoft has recently announced that they will turn over the control of the future direction of Active X to a third party standards committee, and will develop reference implementations for other platforms. With Microsoft's market dominance, it would be foolish to ignore Active X as a vehicle to package our imaging system software.

4.6.1 Component Object Model - Active X

The biggest advantage of using system level objects is the ability to reuse software in a much more standard way. It encapsulates both the data and operations into an

[3]OLE stands for Object Linking and Embedding, but OLE has evolved to be far more encompassing than just Object Linking and Embedding.

object. One could argue that this is nothing special, since programmers have traditionally been able to reuse software in the form of software libraries. This is certainly true — however, these methods of code reuse are language and platform dependent. The standardization of the interface to the objects is the key to these software components. It allows containers to be built that will work with any object (present and future) that conform to the standard. Basically this transforms the interface of software components from a many-to-many mapping to a one-to-one mapping. All the programmer has to worry about is that the container conforms to the standard interface for containers, and that the component conforms to the interface for components. It does not matter what programming language was used to implement the objects because COM is a binary standard.

There has been a lot of interest in Active X for use with Internet Explorer, Microsoft's web browser. Note that Active X is not only for Internet Explorer, it can be used in many other products that support the standard such as Visual Basic, Visual C++, Excel, Word, Access, etc. We will implement a limited version of the panoramic transform in order to demonstrate the potential benefits of leveraging other technologies such as Active X.

Chapter 5

Implementation

5.1 Speed Considerations

The decision to use Dynamic Linked Libraries (DLLs) is usually determined by the intended purpose or functionality to be provided by the DLL. There is very little execution speed penalty associated with the use of DLLs. However, there could be major memory savings if more than one application uses the same set of DLLs, since only one copy of the code will be loaded into computer memory (each application that uses the DLL will have its own data segment). The JPEG and GIF image reading objects were implemented in a DLL.

Another consideration is transparent hardware support. In other words, we want to take advantage of dedicated graphics hardware if it is present, but we will emulate the functionality in software if the hardware is not present. This raises an important issue when it comes to implementation. The issue is whether to optimize specifically for a software solution, or to sacrifice some speed and space optimization in order to develop a solution that can take advantage of the presence of 3D graphics hardware. We choose not to optimize specifically for software, because it is expected that 3D graphics hardware will be common on most PCs in the near future.

One of the techniques for increasing execution speed in computation intensive tasks is to use look up tables (LUTs). Depending on the particular situation, this could trade off computation cycles at the expense of extra memory use. The use of LUTs reduces the computation to constant time. However, not all problems can take advantage of this technique. For this particular project, the cylindrical panoramic transform could use this technique, but the spherical transform could not. The warp-

ing of cylindrical panoramic images is invariant as the virtual camera pans. The spherical transform allows the virtual camera to both pan and tilt, resulting in a spatially variant LUT. This project does not use LUTs to transform the images for several reasons. The primary one is that the LUT method involves the CPU and cannot take advantage of 3D graphics hardware for texture mapping. If 3D graphics hardware was present on a system, then most of the image transformation operations could be done in the graphics hardware, freeing the CPU for other tasks. If LUTs were used, this would not be the case; the 3D hardware would be doing very little, while the CPU would be required to do all the re-mapping operations involving the LUT.

Integer operations are usually faster than floating point operations. On most machines, integer operations take far fewer clock cycles to complete than the equivalent floating point operation. Fixed point numbers can be implemented that use only integer operations, or the ALU (Arithmetic Logic Unit). Fixed point numbers consist of an integer and a fractional part. For instance, a fixed point number could be represented using 2 bytes for the whole number portion, and 2 bytes for the fractional portion. The primary reason for considering fixed point operations over floating point is the speed improvement. The tradeoff is the smaller range of values that can be represented, and less precision. Neither of these limitations is of major significance in computer graphics. The details of the C++ fixed point number class implementation is in the appendix. When the fixed point C++ class was benchmarked informally, it was found that on a 486DX33 PC most operations were roughly twice as fast as the floating point counterpart. However, there were no significant speed improvements when the same benchmarks were performed on a 100MHz Pentium PC. One possible explanation for this is because of load balancing on the CPU. The design of modern CPUs enables most subsystems to execute in parallel if there are no data dependencies. This strategy tries to keep as many subsystems on the CPU busy as possible. It is possible that by using fixed point operations, we are keeping the ALU busy but not the floating point unit (FPU). Thus on a CPU such as the Pentium, the performance gains due to faster fixed point operations are reduced because of less efficient use of the CPU, or less parallelism. It is the author's opinion that the question of whether to use fixed point arithmetic or floating point is dependent on the CPU in question.

Ideally there would be two versions of the implementation and, based on benchmarks during the installation phase, the faster routines would be installed. We leave it up to the graphics engine to make such optimizations.

5.2 Memory Considerations

There are a few issues that influence the runtime memory requirements. One issue is whether to use static libraries or dynamic link libraries (DLLs). With DLLs, the executable (binary) is smaller, but will need to locate and link with the DLLs at runtime. This could cause problems if the user relocates the DLLs to a place that is not in the search path, or deletes the DLLs altogether. Since many applications are built using MFC, it is very likely that using DLLs would save memory at runtime, not to mention saving disk space.

The document/view model, in theory, would save a great deal of memory since there is only one copy of the data (located in the document), and multiple views. In this particular case, due to the fact that the graphics engine does not support multiple cameras, every view must instantiate a graphics engine, thereby creating multiple copies of the input texture images. This limitation should be eliminated when the Intel 3DR graphics engine supports multiple cameras. In practice, it is not a major problem since the operating system manages virtual memory and will swap if necessary. Things are just slowed down a little if there are too many views on the screen, and limited main memory.

The use of quadratic patch gives us a major memory savings compared to the other method of supplying a 3D mesh model. Only nine control points are necessary compared to many vertices and edges required for a mesh.

5.3 Debugging

Debugging is challenging under even the best of circumstances. Many programmers consider this part of software development an art form. Debugging is the process of uncovering and correcting errors that manifest themselves as symptoms in the testing phase. Debugging is *not* testing, rather it occurs as a consequence of testing [20]. Thielen[26] comments on where bugs come from:

> How do bugs get into a program? Very simply, you [the programmer] put them there. You are the one who sits down and types some buggy code into your program... Every time you code, you insert bugs. This is one of the dirty little secrets of programming. Programmers write code and create bugs every day.

5.3.1 Assert the world

It is helpful to use diagnostic functions and macros to ensure that the code itself detects and reports abnormal program states. This functionality is implemented in a large part by the MFC application framework, and the concepts are used elsewhere in the program. One of the more useful practices is to use ASSERT macros. The following code excerpt illustrates the use of the ASSERT macro, in which the *Graphics Context* object is ASSERTed to be valid before any further initialization occurs.

```
void CSphereView::InitializeGC()
{
        ASSERT(m_GC.IsValid());
        if(!m_GC.IsValid())
                return;
        //setup camera
        m_camera.Location(0.0f, 0.0f, 0.0f);
        m_camera.Direction(0.0f, 0.0f, 1.0f); // +z axis
        m_camera.Up(0.0f, 1.0f, 0.0f); // +y axis
        ...
};
```

In programs that use ASSERTs, many errors can be caught and localized very quickly and easily. In order for this scheme to work, the ASSERTs must be used liberally with pre and post conditions being ASSERTed. This in effect is building self diagnostic code. From the software engineering point of view, this can be seen as doing both black box and white box testing on the methods (in the case of objects) and procedures. By using self diagnostic validity checks on the objects themselves, the program can check to ensure that the object is in a valid state.

5.3.2 Visual Aids for Debugging

One of the major problems with using debuggers is the fact that the programmer has to interpret the values of variables. In certain instances it is not possible to specify

conditional breakpoints; this leads to a problem of trying to check and analyze too much information in the hopes of finding the cause of the bug. To further complicate matters, there is the phenomenon called the *probe effect* which stated simply, is that the act of observing changes the observation. This can make certain problems difficult, if not impossible, to debug using the traditional method of setting breakpoints — since the very act of transferring control to the debugger can disturb the effect that you were trying to observe. An example of this would be when trying to debug code that need to be drawn to the screen; when the debugger gains control again at a breakpoint the painting of the client area will not occur correctly. One way to minimize the probe effect would be to refrain from setting breakpoints and just use the time-honoured tradition of printing text based information to the *stderr* or *stdout* streams. This approach has two problems. First, it involves potentially massive amounts of data in textual format that the programmer must sieve through in trying to reconstruct the internal state of the program, followed by analysis of the states to determine the potential causes of the problem. Second, in the Windows 95/NT environments, printing textual information to a stream is not trivial since there are no *stdout* or *stderr* streams. A way is needed to condense the state information and to graphically represent the program state of interest such that, when debugging, the probe effect is minimal and many program states are displayed in graphical format so that the program can execute at near normal speed.

Debugging Bubbles

Debugging bubbles is one such visual debugging aid that was incorporated in the software. This is a novel way of interactively viewing the internal state of the warping algorithm. The bubbles in Figure 5.1 are of varying sizes so that the programmer can distinguish which of the nine control points they are. These points are the quad patch control points in the texture space overlayed in the viewport. In this way, the programmer can view the output of the viewport and the corresponding texture patch. This little bit of extra coding allowed the programmer to visualize the internal states of the program while avoiding the probe effect. For instance, it was helpful in tracking down the discontinuity at the seam of the panospheric image. As the camera crossed the seam, the control point was mapped to a different region which,

in texture space, corresponded to a large jump. Without the visual aid, some of these problems would have been much more difficult and time consuming to diagnose. In Figure 5.1 the right image shows the quad patch control points in texture space projected on to the viewport window. The bubble in the left view in the upper left corner indicates that the program recognizes that the quadpatch crosses a seam. The interesting thing to note here is the control point that is circled free hand. It shows that the distortion that is seen is caused by one of the control points being out of place. Seeing where the control point was actually mapped to, and still being able to interact with the program at full speed, made it relatively simple to diagnose the problem. The reader may have noticed that having the implementation based on the document/view model, may make it possible to create a view that would map the texture points in texture space, that is to say, overlay these control points in the texture image. This would complicate the implementation too much and may introduce errors which would ultimately be self defeating as a debugging aid.

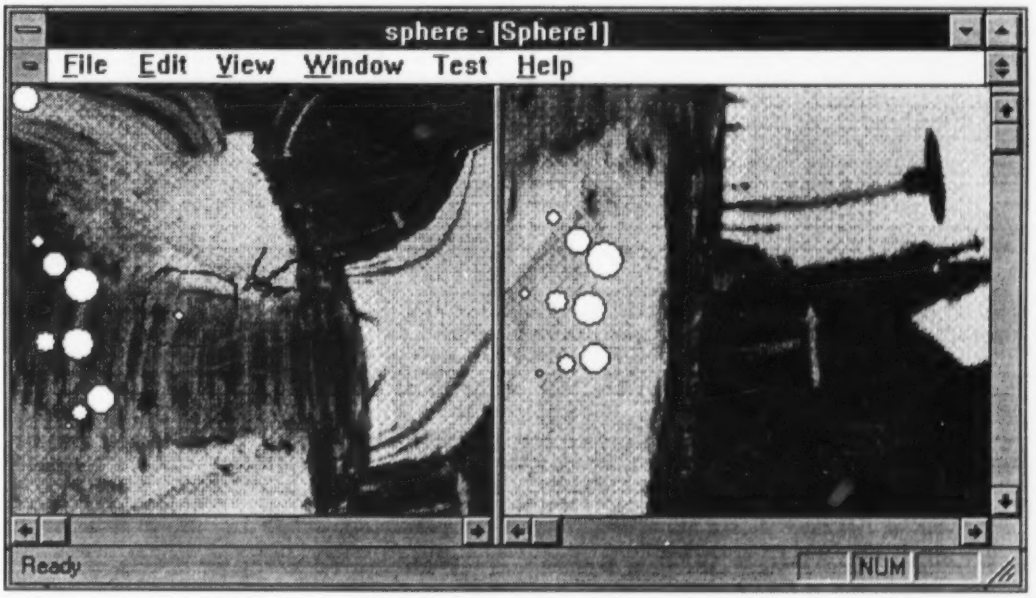


Figure 5.1: Visual Aid for Debugging

5.4 Matching Image Seams

When dealing with multiple images that must be texture mapped to a single 3D model, the major problem that we will encounter is trying to match up the seams. The calculations must be performed with a high degree of precision in order to ensure that the seams are not noticeable if the image is to be stitched together at runtime, as was attempted. The solution is, of course, not to have the seams in the first place. The panospheric image is easier to map because even though there is a seam, the two regions are on the same image so we need not worry about skew or relative rotation of the two regions.

Chapter 6

Results

The interactive aspects of the software cannot be demonstrated on paper, but we can show some screen-shots that display the unwarped images. One of the objectives was to target both Windows 95 and Windows NT 3.51 (or later). Most of the screen-shots were taken when running under Windows NT 3.51 because it was also the development platform.

6.1 Panoramic Images

The image in Figure 6.1 was used to illustrate the distortion correction that the software performs. Notice that the straight vertical lines are still straight in the virtual camera views (Figure 6.2), but the horizontal lines are now curved due to perspective foreshortening. The amount of curvature required to correct for the distortion is dependent on the field of view. As expected, the smaller the field of view the less correction is required and the straighter the horizontal lines become. Figure 6.3 is a sample panoramic image and Figure 6.4 is a view produced by a virtual camera correcting the panoramic distortion.

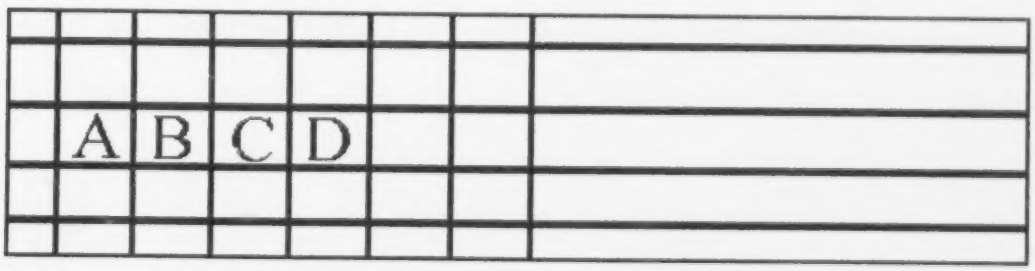
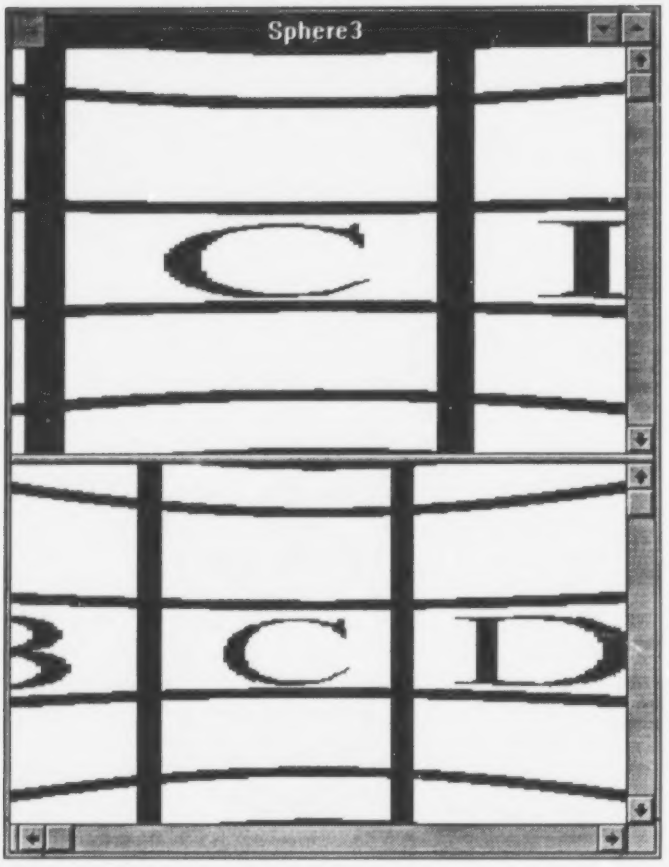


Figure 6.1: Input Panoramic Image



Top View: 40 degree Horizontal field of View
Bottom View:60 degree Horizontal field of View

Figure 6.2: Virtual Camera View of Test Panoramic Image

6.2 Front and Back Fish-eye Image Pairs

Figure 6.5 shows four independent views of the same scene composed of the two fish-eye images. Each of the views can be manipulated by panning, tilting, and zooming the virtual camera.

6.3 Input Image Resolution

The input image resolution dictates, to a large extent, the quality that can be expected when scenes are viewed with the virtual camera. Since the views are usually

Figure 6.3: Input Panoramic Image

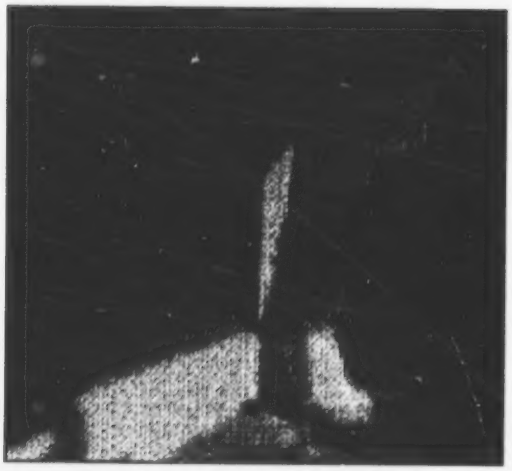
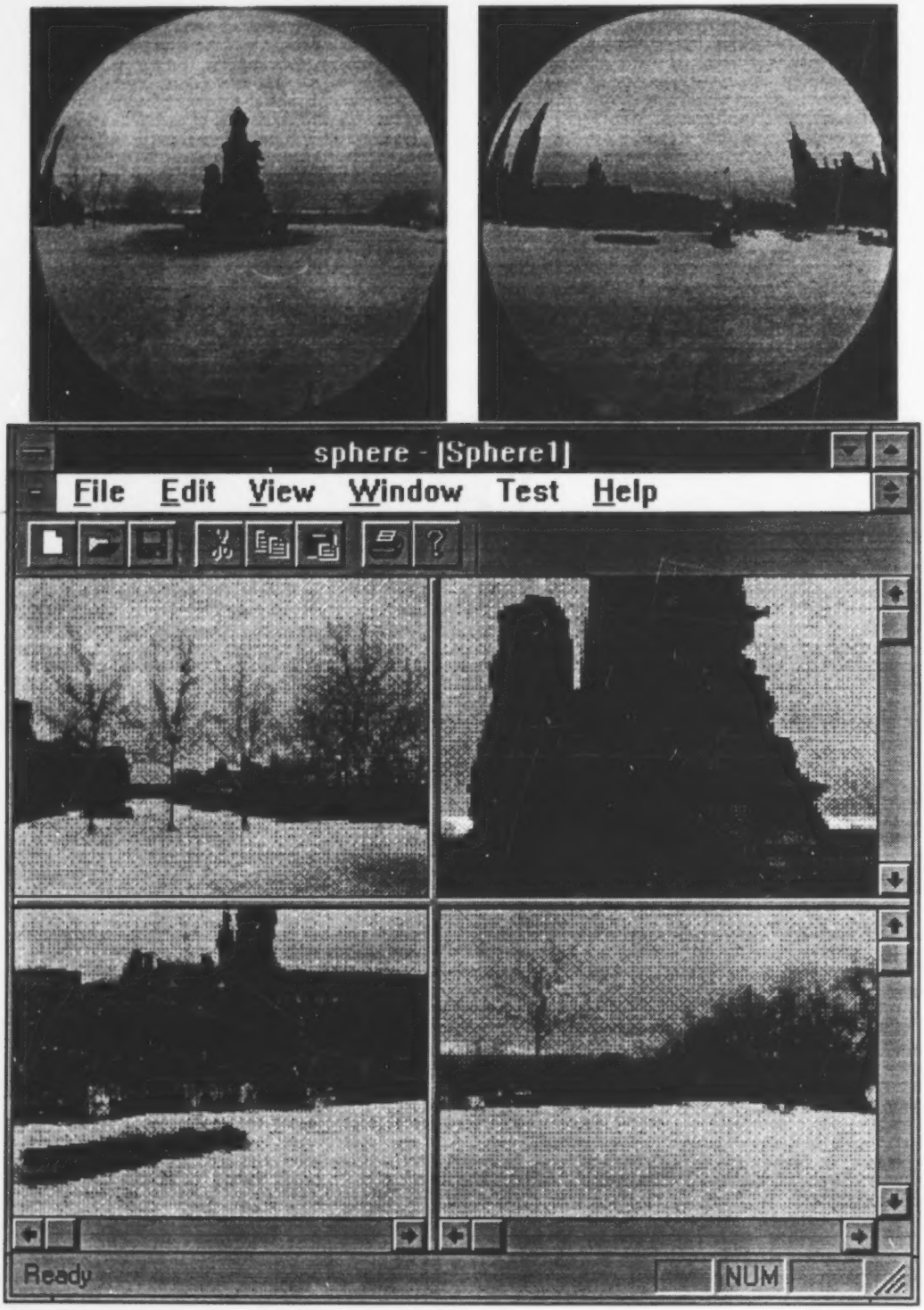


Figure 6.4: Virtual Camera View of Panoramic Image



Top Row: Fish-eye image pair
Bottom: Four independent views of the same scene

Figure 6.5: Views of Front and Back Fish-eye images

of relatively small portions of the input image, pixelation can be very apparent. Figure 6.6 shows two views of the same scene, but the source images are of different resolution. The input image with higher resolution will produce better results. Although this may be obvious, the problem is more apparent because each texel is mapped to many pixels on the screen, thus reducing the natural low-pass filtering that occurs on monitors.

6.4 Panospheric Image Mapped to a Panoramic Image

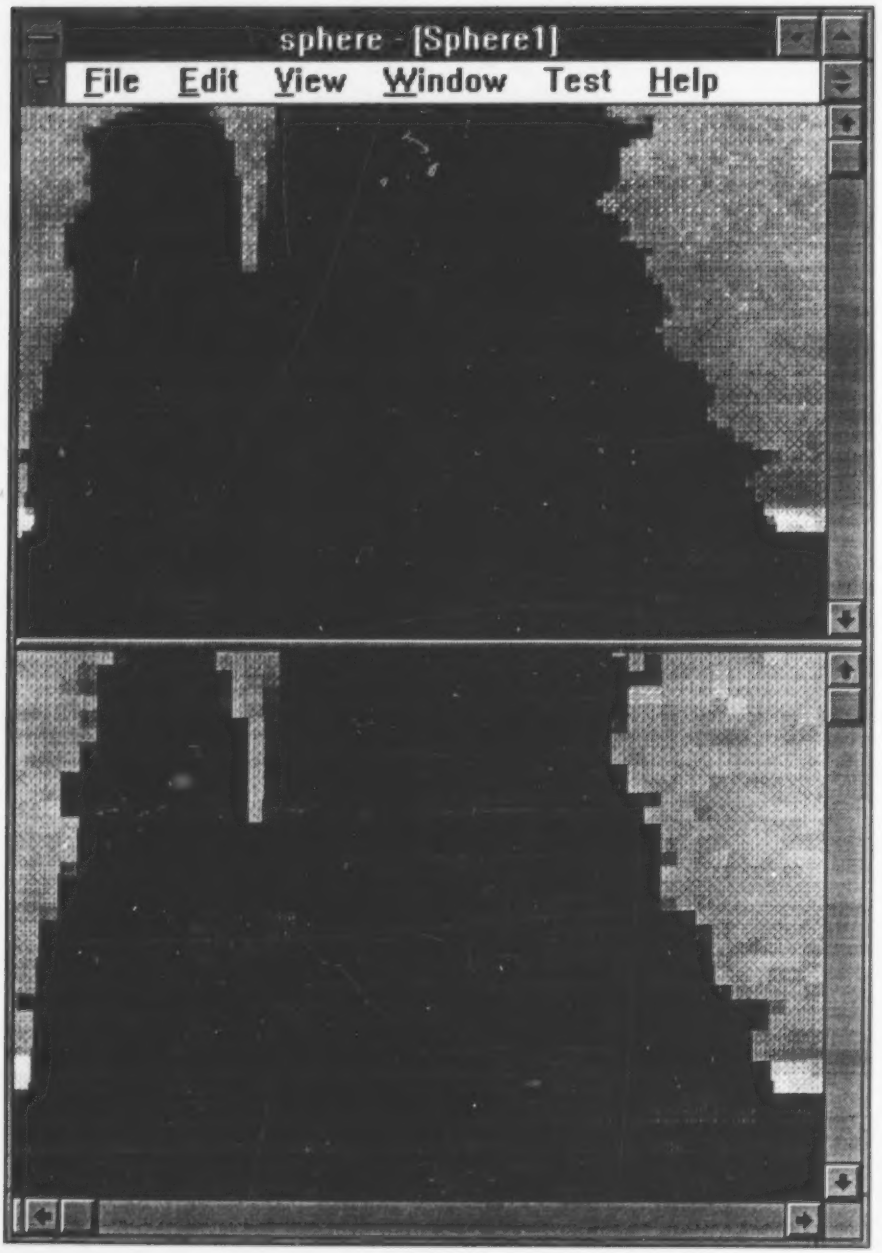
Figure 6.7 is a view of a panospheric image transformed to a panoramic image. The virtual camera supports panning to the left or right. The source panospheric image is given in Figure 3.12.

6.5 Windows 95/NT

Figure 6.8 is a screen-shot of the program running under the Windows 95 environment. Most of the other screen-shots where taken under Windows NT 3.51. This is included to show that the goal of designing a program for both operating systems is satisfied. It also illustrates a view of the panospheric transform (left window).

6.6 Image Compression

The images in Figure 6.9 demonstrate the impact of using high compression ratio on the size and quality of the final unwarped images. The original image is displayed in the upper left corner; the other three images are screen-shots of the unwarped conventional camera view. All the source images were 24 bit 1024x1024 images, each compressed to at least the compression ratio shown. Note that the checker patterns are not present in the screen display of the images; they were introduced in the conversion process to get it on paper. It may not be as apparent in the figure on paper, but on the screen we can notice very little difference between the results of the virtual camera viewing the 40:1 compressed image or the 55:1 compressed image. The quality degraded, however, when viewing the 80:1 compressed image, including



Images viewed using a virtual camera
Top: 512x512 fish-eye source image **Bottom:** 256x256 fish-eye source image

Figure 6.6: Different Resolution Source Images

Figure 6.7: Panospheric Image Mapped to a Panoramic Image

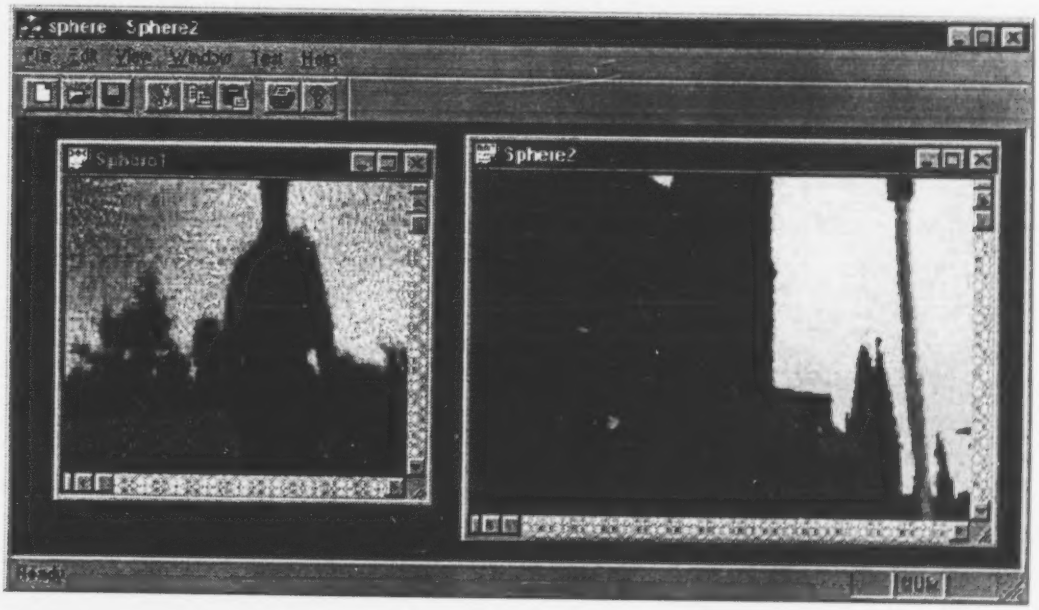
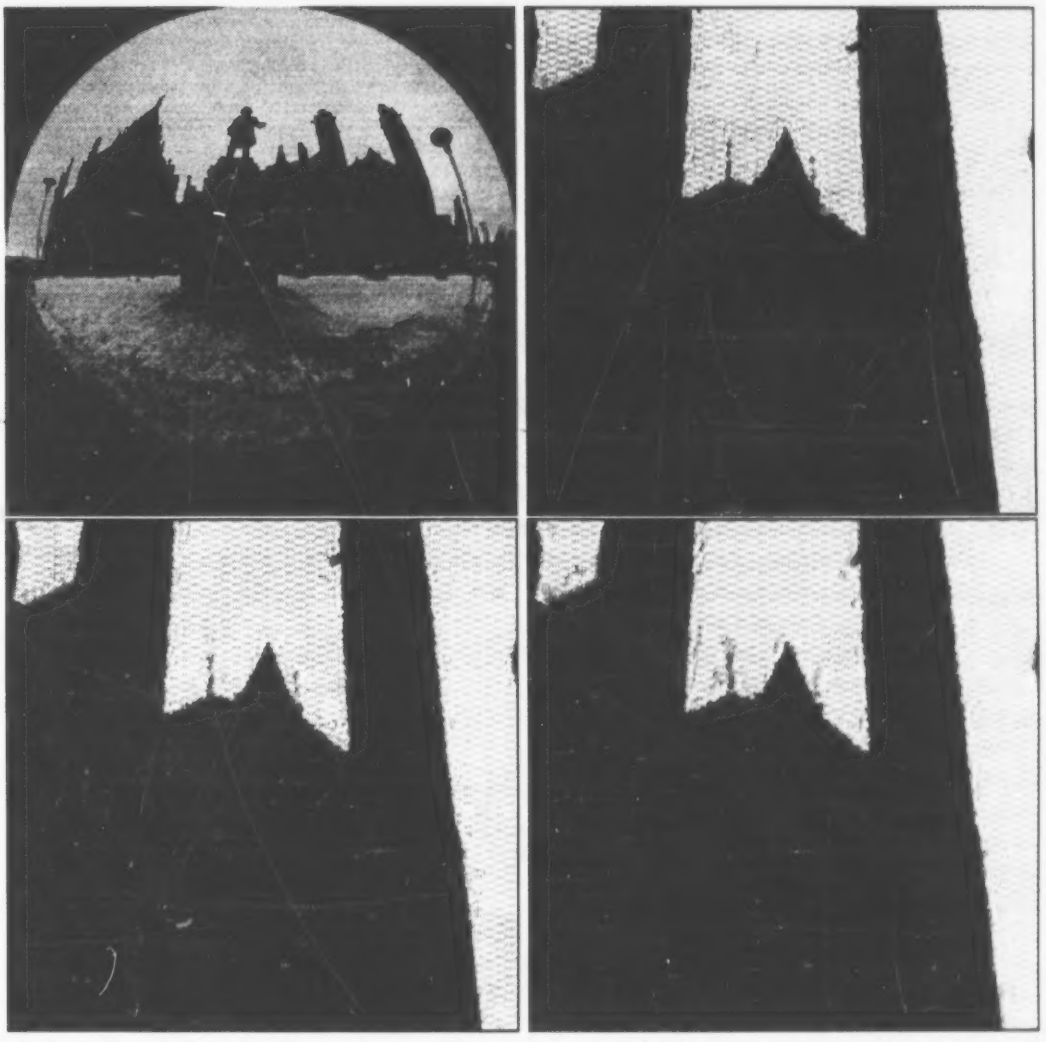


Figure 6.8: Windows 95 User Interface

loss of color information for the edges of the roof. Uneven color loss resulted because JPEG quantized some of the 8x8 sub-blocks color information out of existence. This only occurs when high compression ratios are involved. This illustrates that we can compress the source images to a fairly small size and still get reasonable quality, even after the virtual camera viewing transform. For instance, the original uncompressed image was 1024x1024x24 bits or 3 megabytes; the largest of the three compressed source images was less than 80 kilobytes. However, these compression ratios are dependent on the content of the images.

6.7 Active X control

The screen-shot in Figure 6.10 is of an Active X control that was quickly ported from the existing application code. The Active X control is embedded in an HTML (HyperText Markup Language) page. Since the control was developed to demonstrate the potential of using Active X with this imaging technology, only the basic functionality of a panoramic transform was implemented. The web page is composed of other technologies such as JavaScript, which is used to connect the objects on that page. All the buttons were programmed with JavaScript to send messages to the Active X object. For instance, when the **Load File** button is pressed, it retrieves the file name from the textedit box above, and sends it to the panoramic Active X object. The important thing to note here is that we can leverage technologies such as Active X to open up many possibilities for this immersive imaging technology.



Top left: Original fish-eye image **Top right:** virtual camera view of compressed 40:1 source image
Bottom left:virtual camera view of compressed 55:1 source image
Bottom right:virtual camera view of compressed 80:1 source image

Figure 6.9: Impact of JPEG Compressed Source Images

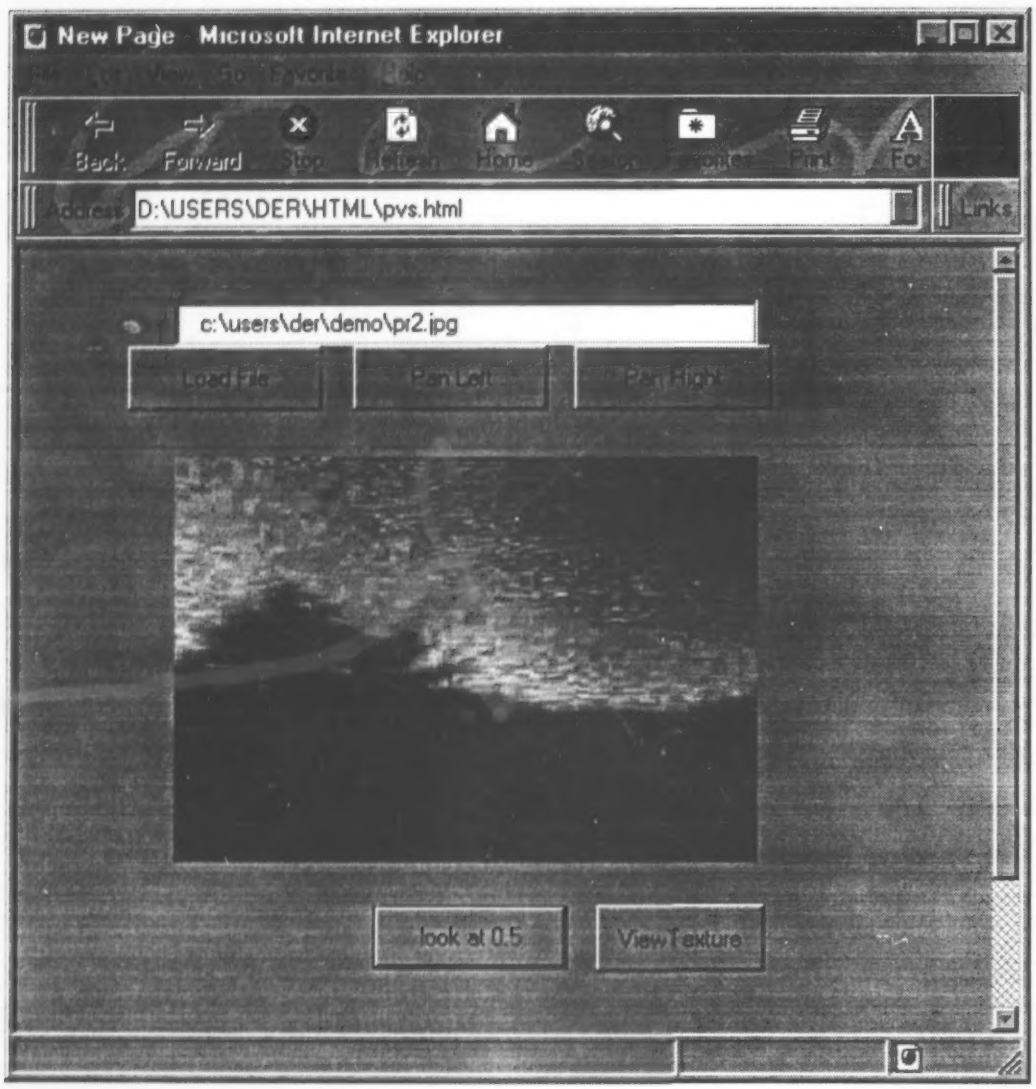


Figure 6.10: Screen-shot of Internet Explorer with a panoramic imaging Active X control

Chapter 7

Discussion

7.1 Performance Evaluation

There will be very little performance analysis given in this thesis. The reason is that there are too many variables that affect the performance of the software system as a whole. The configurations of PCs are highly variable. Performance could be affected by the CPU model and speed, the bus speed, the bus architecture, the BIOS settings, the amount of main memory, the type of memory, the amount of memory on the graphics card, the type of graphics card, the operating system used, the version of the device drivers being used, the window size, the number of windows open, the background services running — even the way the user moves the mouse, to name just a few of the factors. Measurements and timings would not be very meaningful.

The *bottleneck*, or performance limiting factor, varies from system to system and configuration to configuration (on the same system). The major factor in the absence of 3D graphics hardware is the speed of the CPU. The other major factor is the performance of the graphics card. Some graphics cards allow fast block transfers which facilitate smooth animation and increased frame rates.

7.2 Advantages of Panospheric Imaging

There are many advantages to using the panospheric imaging system over others.

- Less expensive to use due to the fact that the panospheric optic can be mounted on regular video and photographic cameras. The panoramic camera and fish-eye lens systems are not mass produced thus the prices are relatively high.

- Easier image capture — Omniview's solution require calibrated fish-eye image pairs, while the panospheric optic needs only a single photograph. However, the panospheric optic does have a blind-spot in the cone where the camera itself occludes the view. But, this blind-spot is relatively small, and greater than 85% field is achievable.

- Less preprocessing required to take the raw photographs and convert it to a form usable by the viewing program. Omniview and Apple's QuickTime VR both use stitching to compose the input images.

- Preprocessing panospheric images is less computationally intensive due to the fact that the field of view is captured on a single imaging plane. Even though there is a discontinuity between the reflected field and the lens field, there is no relative rotation between the two regions. With this constraint, the stitching process is simplified computationally.

- The panospheric sensor can be used to capture live video making interactive panospheric movies a possibility for the future. The revolving panoramic camera can't be used to capture moving objects because at any instant in time, it can't see a full 360° field of view.

- A variant stereo panospheric optic can be used to extract depth information of objects in a scene. This possibility is not present in the other systems.

- Features Unique Our System

  - Based on the document/view model which facilitates the use of multiple independently controlled virtual cameras viewing a single scene.

  - Multiple viewing models for the various formats, such as virtual conventional camera which is planar and panoramic strips which is non-planar.

  - supports cylindrical, spherical, and panospheric images. No other system supports all three models — They either support the cylindrical model or the spherical model. This is the only system that also supports the panospheric format.

- used object oriented technology such as Active X to allow tighter integration with other applications.

7.3 Future Work

7.3.1 Panospheric Video

Panospheric video would allow the user not only to look around from a stationary position, but also to move to different locations. There are several scenarios to consider. One scenario is to capture a sequence of static images and make this into a video stream. Conceivably, this could be used for applications such as viewing of real estate, in which panospheric images are taken in each room and the user is then able to navigate to different rooms. In this case, the video stream is just a convenient way of holding a collection of panospheric images; each frame could be viewed in any sequence. Another scenario is where the panospheric video is captured continuously as the camera moves in a scene. The user would be restricted to the path that the camera took, but at any point in time they would be able to look around from that given spot.

There are many formats that will encode and compress video streams. MPEG is a commonly used standard for compressing video. Windows AVI is another standard — which allows the user to specify the compression codec[1]. The panospheric transformation would not change but, instead of using a single image as the texture map, we would use an image in a frame of video. There are a few factors that must be considered. If compression is to be used, unless there is hardware support for decompression and/or 3D graphics, the processor may not be able to keep up with decompressing the video stream and rendering the distortion corrected view. If we use uncompressed video to off load the processing burden, the storage requirements will increase dramatically. These problems exist only in the short term. It is only a matter of time before video decompression and 3D graphics hardware become affordable for PCs, enabling the potential of panospheric video to be realized for the mass market. A third scenario is the use of live panospheric video. In this case, the panospheric images would be continuously captured using a framegrabber. Again,

[1]Codec is short for coder/decoder

each frame would be used as the texture image in the transforms. One could conceivably locate different virtual camera on different computers over a relatively fast network, allowing each user to look in the direction they want. Panospheric movies would be a good application of this technology.

7.3.2 Automation of Preprocessing

In the current implementation some of the preprocessing steps are done manually. When photographs are scanned into the computer the image must be manually cropped. Panospheric images are circular, thus we must crop the image to the minimum rectangular region that encloses the active region. This step can conceivably be done automatically by using image processing techniques to detect the boundaries of the active circular region, and cropping accordingly. We could take advantage of our knowledge of the content of the image and do edge detection on select rows.

7.3.3 Integration with other Applications

In order for this technology to be widely used and accepted by consumers, it must be able to integrate with other technologies and applications that are currently in use — such as the Netscape web browser and Microsoft Office applications. There are many ways to integrate this product with other applications. One is to develop applications that explicitly support other specific applications; another way is to use a technology such as COM (Component Object Model). COM is a binary standard for system level objects. It allows developers to package their software components and have other applications use it. The other applications do not need any specific information about the components. Objects themselves will expose the operations that can be performed on them. As has been discussed earlier, Active X is built on top of COM and can be used to facilitate integration with other applications. Although a sample Active X control was implemented to demonstrate the potential, a lot of work still need to be done to make the control robust and network-capable.

Navigational and data links can be embedded in the panospheric images to enable the user to navigate to other scenes. These links could include Uniform Resource Locators (URLs) and connect to other data resources such as the World Wide Web (WWW). Imagine looking around in a panospheric scene of a museum: you see an

interesting object and want a closer look, so you click on the object and the web browser retrieves more information, be it static pictures, videos or sounds.

7.3.4 Head Mounted Display

Head Mounted Displays (HMD) with head tracking can be used so that as the user moves their head, the panospheric scene is updated to reflect the direction in which they are looking. HMDs in conjunction with the panospheric video streams would provide a better simulation, since this method offers the user more freedom than just using a single panospheric image. With the use of HMDs, the prospects of stereo panospheric imaging can also be explored.

7.3.5 Internet Applications

To fully exploit the Internet we can use technologies such as Active X, which is gaining a large following and has the backing of Microsoft, or we can consider using Java to write applets which are secure and platform independent. Active X at the present time is system dependent, although this may change in the future. As network bandwidth increases for the typical user, this imaging technology may become very popular.

Chapter 8

Conclusion

Immersive imaging technology is a rapidly growing field with many potential applications. The ability to capture panospheric images with the use of inexpensive conic mirrors, coupled with the ability to simulate interactive viewing with a virtual camera in software, will enable multimedia content creators to present virtual reality in a way not possible with computer generated graphics or static photographs alone. A software program was developed that enables the user to view immersive images of real world scenes. The software is based on the document/view model, which facilitates the viewing of the same scene with multiple virtual cameras. The software supports common image file formats, such as JPEG and GIF, which provide varying levels of compression. The compressed source images save disk space, and if transferred over networks would require far less bandwidth. Many immersive image formats were supported, including panoramic images which are modeled as cylinders, panospheric images which are modeled as spheres, and fish-eye image pairs which are also modeled as spheres. Due to the non-planar projections involved in the capturing process, the input images are warped. Our software maps these warped images onto the appropriate 3D models, in effect reconstructing the scene, and then it resamples the scene by viewing the model with a virtual camera. When the images (mapped onto 3D models) are viewed with the virtual cameras, the distortions are eliminated. The virtual cameras can be manipulated using pan, tilt, and zoom operations, thereby allowing the users to interactively look around as if they were immersed in the scene.

The major advantage of using the PVSI's conic mirror panospheric system is the ease with which images are captured. It is far more economical than fish-eye lenses.

Even though the panospheric images have a discontinuity or a seam, it is relatively easy to stitch the two regions together, since there is no relative rotation between the two regions. With the use of PVSI's relatively inexpensive panospheric image capture system and this software designed to be run on home computers, the potential mass market applications have yet to be explored. Immersive imaging is an enabling technology that will be the cornerstone of many future multimedia applications.

Bibliography

[1] A. Basu. Personal Communication.

[2] S. Bogner. Personal Communication.

[3] S. L. Bogner. An Introduction to Panospheric Imaging. *Proceedings of the 1995 IEEE International Conference on Systems, Man and Cybernetics*, 1995.

[4] E. Catmull and R. Rom. A Class of local interpolating splines. In *Computer Aided Geometric Design.*

[5] S. E. Chen. QuickTime VR - An Image-Based Approach to Virtual Environment Navigation. In *Computer Graphics Proceedings, Annual Conference Series.* Apple Computer, Inc., 1995.

[6] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 2nd edition, 1990.

[7] R. C. Gonzalez and R. E. Woods. *Digital Image Processing.* Addison-Wesley, 1992.

[8] N. Green and P. Heckbert. Creating raster Omnimax images from multiple perspective views using elliptical weighted average filter. *IEEE Computer Graphics and Applications*, pages 21–27, 6 1986.

[9] MPC Working Group. *Multimedia PC Standard.* Software Publishers Association.
URL: http://www.spa.org/mpc

[10] T.R. Halfhill. See You Around. *Byte*, pages 85–90, May 1995.

[11] Intel Corporation. *Intel 3DR Graphics Pipeline Programming Manual*, 1995.

[12] Intel Corporation. *Intel 3DR Rasterizing Engine Programming Manual*, 1995.

[13] M. Irani, S. Hsu, and P. Anandan. Video compression using mosaic representations. *Signal Processing: Image Communication*, v 7(n 4-6):p 529–552, November 1995.

[14] Microsoft. *For Developers Only.*
URL: http://www.microsoft.com/devonly

[15] Microsoft Corporation. *Win32 SDK*, 1985-1995.

[16] Microsoft Corporation. *Windows Interface Design Guidelines for Software Design*, 1995.

[17] D. O'Donovan and P. O'Connor. Multiple Image Reconstitution with Lens Correction. In *Proceedings of SPIE - The International Society for Optical Engineering*, pages 131–141, 1995.

[18] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[19] I. Powell. Panoramic Lens. *Applied Optics*, Vol. 33(No. 31):p 7356–7361, November 1994.

[20] R.S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, 3rd edition, 1992.

[21] J. Reyda. Personal Communication.

[22] D. Southwell. Personal Communication.

[23] D. Southwell, B. Vandegriend, and A. Basu. A Conical Mirror Pipeline Inspection System. In *Proceedings of the 1996 IEEE, International Conference on Robotics and Automation*, April 1996.

[24] Sun Microsystems Inc. *Java(TM) Developers Corner*.
URL: http://java.sun.com/devcorner.html

[25] R. Szeliski. Image Mosaicing for Tele-Reality Applications. In *Proceedings of the 2nd IEEE Workshop on Applications of Computer Vision*, pages 44–53, 1994.

[26] D. Thielen. *No Bugs! Delivering Error-Free Code in C and C++*. Addison-Wesley, 1992.

[27] N. Trevett. Simulation of the Real World. *Computer Systems Europe*, 10(11):25–27, November 1990.

[28] Usenet. C++ FAQ.
URL: http://www.cis.ohio-state.edu/hypertext/faq/usenet/FAQ-List.html

[29] Usenet. Compression FAQ.
URL: http://www.cis.ohio-state.edu/hypertext/faq/usenet/FAQ-List.html

[30] Usenet. FAQ: 3-D Information for the Programmer.
URL: http://www.cis.ohio-state.edu/hypertext/faq/usenet/FAQ-List.html

[31] Usenet. Graphics File Format FAQ.
URL: http://www.cis.ohio-state.edu/hypertext/faq/usenet/FAQ-List.html

[32] Usenet. JPEG Image Compression: Frequently Asked Questions.
URL: http://www.cis.ohio-state.edu/hypertext/faq/usenet/FAQ-List.html

[33] Y. Yagi, Y. Nishizawa, and Y. Masahiko. Estimation of Free Space for the Mobile Robot using Omnidirectional Image Sensor COPIS. In *Proceedings of the 1991 International Conference on Industrial Electronics, Control and Instrumentation - IECON'91*, 1991.

[34] K. Yamazawa, Y. Yagi, and M. Yachida. Obstacle detection with omnidirectional image sensor HyperOmni vision. In *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, 1995.

Appendix A

C++ Fixed Point Number Class

```
#ifndef FIXEDPT_H
#define FIXEDPT_H

#include <iostream.h>
//============================================================================
// (C)1995 Kenneth P. Der (der@cs.ualberta.ca)
//============================================================================
// Contents:
//      This file implements a template class for fixedpoint numbers. Fixed
//      point numbers should be slightly faster than floating point numbers
//      but have a much more limited range and accuracy.
//----------------------------------------------------------------------------
// class specific method naming convention:
//      {l|m|s}{l|m|s}{Mult|Div}[Assign](const FixedPt<T,UT,nFPt> &rc2)
//      sxMultAssign() -- small *= unknown range
//      llDivAssign() -- large /= large
//
//      [*][x][x][x] -- large
//      [0][*][x][x] -- medium
//      [0][0][*][x] -- small
// Caution:
//      -since we're using 2's complement according to the above classification
//       the more negative numbers (e.g. -5000.0 ) are classified as "small"
//       and the less negative numbers (e.g. -0.897 ) are classified as "large"
//      -don't use methods involving 's' classification if you can avoid it --
//       unless you know EXACTLY what going on -- or else you'll get incorrect
//       results.
//              ie, avoid ls,ms,ss,sl,sm methods.
//      -of all the Mult methods, xx is the most accurate but also the slowest.
//
// Note:
//   use multiples of 2 for nFPt
//   The suggested value for nFPt is sizeof(T)*4
//       i.e. if T is 32 bits then nFPt is 16
// Known Bugs/problems: search for [kpd]
//      - msvc++2.0 doesn't like friend functions in template classes.
//      - thus stream insertion/extraction ("<<" and ">>") operators are not
//        supported at this time.
//      - a workaround is to use the cast operators to convert to the intrinsic
//        types first.
//      - or write stream operators for each template class (i.e. each instance
//        of the class template) yourself
//
// T - is the integer type used for the internal representation
// UT - is the unsigned T type used to make sure the signs are correct of the
//      bitshifts
//============================================================================
template <class T, class UT,int nFPt>
class FixedPt
```

```
{
public:
        // ctor dtor
        FixedPt(float value) { m_fixedPoint = T(value * (1<<nFPt));}
        FixedPt(double value){ m_fixedPoint = T(value * (1<<nFPt));}
        FixedPt(int value=0)   { m_fixedPoint = value << nFPt;}
        FixedPt(const FixedPt<T,UT,nFPt> &rc){ m_fixedPoint = rc.m_fixedPoint;}
        // virtual ~FixedPt(){}; //hmmm... will this cause a vtable to be created?

        // mutative operators
        FixedPt<T,UT,nFPt> & operator += (const FixedPt<T,UT,nFPt> &rc2)
                { m_fixedPoint += rc2.m_fixedPoint; return *this;}
        FixedPt<T,UT,nFPt> & operator -= (const FixedPt<T,UT,nFPt> &rc2)
                { m_fixedPoint -= rc2.m_fixedPoint; return *this;}

        FixedPt<T,UT,nFPt> & llMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (m_fixedPoint>>(nFPt>>1))*(rc2.m_fixedPoint >>( nFPt>>1));
                        return *this;
                }
        FixedPt<T,UT,nFPt> & lmMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (m_fixedPoint>>(nFPt>>1))*(rc2.m_fixedPoint >>( nFPt>>1));
                        return *this;
                }
        FixedPt<T,UT,nFPt> & lsMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (m_fixedPoint>>nFPt)*rc2.m_fixedPoint;
                        return *this;
                }
        FixedPt<T,UT,nFPt> & mlMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (m_fixedPoint>>(nFPt>>1))*(rc2.m_fixedPoint >>( nFPt>>1));
                        return *this;
                }
        FixedPt<T,UT,nFPt> & mmMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (m_fixedPoint>>(nFPt>>1))*(rc2.m_fixedPoint >>( nFPt>>1));
                        return *this;
                }
        FixedPt<T,UT,nFPt> & msMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = ((m_fixedPoint>>(nFPt>>1))*rc2.m_fixedPoint)>> (nFPt>>1);
                        return *this;
                }
        FixedPt<T,UT,nFPt> & slMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (m_fixedPoint)*(rc2.m_fixedPoint >> nFPt);
                        return *this;
                }
        FixedPt<T,UT,nFPt> & smMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (m_fixedPoint>>(nFPt>>1))*(rc2.m_fixedPoint >>( nFPt>>1));
                        return *this;
                }
        FixedPt<T,UT,nFPt> & ssMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = (UT)(m_fixedPoint*rc2.m_fixedPoint)>> nFPt;
                        return *this;
                }
        FixedPt<T,UT,nFPt> & xxMultAssign (const FixedPt<T,UT,nFPt> &rc2)
                { // (a.b) * (c.d) --> (a>>nFPt)*(c.d)+ (.b)*(c>>nFPt) + (((.b)*(.d)))>>nFPt)
                        m_fixedPoint = wholeLowWord()*rc2.m_fixedPoint+ // a*(c.d) +
                           fractionLowWord()*rc2.wholeLowWord() +  // (.b)*c
                           (((UT)(fractionLowWord()*rc2.fractionLowWord())))>>nFPt); // (.b)*(.d)
                        return *this;
                }

        FixedPt<T,UT,nFPt> & operator *= (const FixedPt<T,UT,nFPt> &rc2)
                {       return mmMultAssign(rc2);}

//llDiV
        FixedPt<T,UT,nFPt> & llDivAssign(const FixedPt<T,UT,nFPt> &rc2)
                {       m_fixedPoint = m_fixedPoint/(rc2.m_fixedPoint>>nFPt);
                        return *this;
                }
        FixedPt<T,UT,nFPt> & lmDivAssign(const FixedPt<T,UT,nFPt> &rc2)
```

```
        {       m_fixedPoint = (m_fixedPoint/rc2.m_fixedPoint) <<nFPt;
                return *this;
        }
FixedPt<T,UT,nFPt> & lsDivAssign(const FixedPt<T,UT,nFPt> &rc2)
        {       m_fixedPoint = (m_fixedPoint/rc2.m_fixedPoint) <<nFPt;
                return *this;
        }
FixedPt<T,UT,nFPt> & lxDivAssign(const FixedPt<T,UT,nFPt> &rc2)
        {       m_fixedPoint = (m_fixedPoint/rc2.m_fixedPoint) <<nFPt;
                return *this;
        }
FixedPt<T,UT,nFPt> & mxDivAssign(const FixedPt<T,UT,nFPt> &rc2)
        {       m_fixedPoint = ((m_fixedPoint<< (nFPt>>1))/rc2.m_fixedPoint) <<(nFPt>>1);
                return *this;
        }
FixedPt<T,UT,nFPt> & sxDivAssign(const FixedPt<T,UT,nFPt> &rc2)
        {       m_fixedPoint = (m_fixedPoint<<nFPt)/rc2.m_fixedPoint;
                return *this;
        }
FixedPt<T,UT,nFPt> & xxDivAssign(const FixedPt<T,UT,nFPt> &rc2)
        { // (a.b) / (c.d) --> ((a/(c.d))<<nFPt)+ (.b<<nFPt)/(c.d)
                m_fixedPoint = ((wholeHighWord()/rc2.m_fixedPoint) <<nFPt)+
                                fractionHighWord()/rc2.m_fixedPoint;
                return *this;
        }
FixedPt<T,UT,nFPt> & operator /= (const FixedPt<T,UT,nFPt> &rc2)
        {       return mxDivAssign(rc2);}


// non-mutative operators
FixedPt<T,UT,nFPt> operator + (const FixedPt<T,UT,nFPt> &rc2)
        { return FixedPt<T,UT,nFPt>(*this) +=rc2;}
FixedPt<T,UT,nFPt> operator - (const FixedPt<T,UT,nFPt> &rc2)
        { return FixedPt<T,UT,nFPt>(*this) -=rc2;}
FixedPt<T,UT,nFPt> operator * (const FixedPt<T,UT,nFPt> &rc2)
        { return FixedPt<T,UT,nFPt>(*this) *=rc2;}
FixedPt<T,UT,nFPt> operator / (const FixedPt<T,UT,nFPt> &rc2)
        { return FixedPt<T,UT,nFPt>(*this) /=rc2;}
FixedPt<T,UT,nFPt> & llMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).llMult(rc2);}
FixedPt<T,UT,nFPt> & lmMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).lmMult(rc2);}
FixedPt<T,UT,nFPt> & lsMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).lsMult(rc2);}
FixedPt<T,UT,nFPt> & mlMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).mlMult(rc2);}
FixedPt<T,UT,nFPt> & mmMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).mmMult(rc2);}
FixedPt<T,UT,nFPt> & msMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).msMult(rc2);}
FixedPt<T,UT,nFPt> & slMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).slMult(rc2);}
FixedPt<T,UT,nFPt> & smMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).smMult(rc2);}
FixedPt<T,UT,nFPt> & ssMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).ssMult(rc2);}
FixedPt<T,UT,nFPt> & xxMult (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).xxMult(rc2);}

FixedPt<T,UT,nFPt> & lxDiv (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).lxDivAssign(rc2);}
FixedPt<T,UT,nFPt> & mxDiv (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).mxDivAssign(rc2);}
FixedPt<T,UT,nFPt> & sxDiv (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).sxDivAssign(rc2);}
FixedPt<T,UT,nFPt> & xxDiv (const FixedPt<T,UT,nFPt> &rc2)const
        {       return FixedPt<T,UT,nFPt>(*this).xxDivAssign(rc2);}
```

```
        // comparison operators
        //[kpd] - use STL for the other operators defined in terms of "==" and "<"
        int operator == (const FixedPt<T,UT,nFPt> &rc) const
                { return m_fixedPoint == rc.m_fixedPoint;}
        int operator <  (const FixedPt<T,UT,nFPt> &rc) const
                { return m_fixedPoint < rc.m_fixedPoint;}
        // cast operators
        operator float() const{ return float(m_fixedPoint) / (1<<nFPt);}
        operator double()const{ return double(m_fixedPoint) / (1<<nFPt);}
        operator int()const{ return (m_fixedPoint >> nFPt); }
        // non-mutative methods
        T round()const { return (m_fixedPoint + (1<<(nFPt-1))) >> nFPt;}
        T ceiling()const { return (m_fixedPoint -1 + (1<<nFPt)) >> nFPt;}
        // mutative methods
        FixedPt<T,UT,nFPt>& dropWhole(){ m_fixedPoint &= fractionMask(); return *this;}
        FixedPt<T,UT,nFPt>& dropFraction(){ m_fixedPoint &= wholeMask(); return *this;}

        // implementation methods
protected:
        T wholeLowWord()const { return (m_fixedPoint>>nFPt);}
        T wholeHighWord()const { return (m_fixedPoint& wholeMask());}
        T fractionLowWord()const { return (m_fixedPoint & fractionMask());}
        T fractionHighWord()const { return (m_fixedPoint<<nFPt);}
        T wholeMask()const { return (~((T)0)<<nFPt);}
        T fractionMask()const { return ~wholeMask();}
        int negativeResult(T a, T b){ return (a^b)&(1<<(sizeof(T)*8-1)) ?1:0;}
        // data members
protected:
        T m_fixedPoint;
private:

/* //[kpd] msvc++2.0 doesn't like this. I guess it's not too friendly :)
public:
        friend ostream & operator << (ostream &os,const FixedPt<T,UT,nFPt> &rc)
                { return os<<float(rc);}
*/
};

//============================================================================
// common typedefs
//      - we'll assume int is 4 bytes
//----------------------------------------------------------------------------
typedef FixedPt<int,unsigned int,sizeof(int)*4> fixed32;

ostream & operator << (ostream &os, const fixed32 &rc)
{
        return os << float(rc);
};

#endif
```